A Comparison Between SISAL 1.2 and Funcalc

Alexander Asp Bock

IT University Technical Report Series **TR-2019-205**

ISSN 1600–6100 **March 2019**

ISSN 1600–6100

ISBN 978-87-7949-371-1

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web `www.itu.dk`

A Comparison Between SISAL 1.2 and Funcalc

Alexander Asp Bock
Computer Science Department

IT UNIVERSITY OF COPENHAGEN

Table of Contents

1 Introduction

The purpose of this technical report is to evaluate the expressiveness of Funcalc by translating programs written in the Streams and Iteration in a Single Assignment Language (SISAL) programming language to Funcalc using sheet-defined functions (SDFs). We also examine differences in their type system, language constructs and syntax. The report unveils that Funcalc is able to express 16 SISAL programs of varying complexity taken from a tutorial on SISAL [1]. Prior work has also demonstrated Funcalc's expressiveness by translating Excel financial functions to Funcalc [2]. The comparison is one-way from SISAL to Funcalc so we do not concern ourselves with SISAL's ability to express Funcalc SDFs. We highlight cases where it is either difficult or impossible to translate from SISAL to Funcalc.

This report is neither an introduction to SISAL nor Funcalc. We assume that the reader is already adequately familiar with both of them. If not, we refer the reader to resources on Funcalc [2, 3] and SISAL [1, 4].

The report is structured as follows. In section 2, we examine the differences between SISAL and Funcalc in terms of their types, syntax and error handling. In section 3, we present the translation of the 16 SISAL programs from [1] to Funcalc using SDFs. Lastly, in section 4, we conclude the report by summarising our observations on the capability of Funcalc to express SISAL programs, and highlight several directions for future work based on the difficulties encountered during translation of the programs in section 3.

2 Differences Between SISAL and Funcalc

While this report focuses solely on SISAL 1.2, there has been some development on version 2.0 [5] and evidence that SISAL 3.1 has been in development [6].

SISAL is a single-assignment, functional, statically typed language. The SISAL language reference [4] does not mention how memory is handled or any functions for allocating and deallocating memory. Therefore we assume that memory is either garbage collected or that appropriate calls to allocation and deallocation functions are inserted at the proper points in the program. Overall, SISAL's syntax and precedence rules are similar to those of Fortran, Pascal and C.

Funcalc is a higher-order, functional spreadsheet language. There are no explicit types in Funcalc and it mostly resembles a dynamically typed programming language. Memory is garbage collected as the underlying implementation is written in C#.

2.1 Types

In this section, we discuss how SISAL's simple and compound types can be expressed in Funcalc.

2.1.1 Standard Types

SISAL has types for scalar values: `boolean`, `integer`, `real`, `double_real`, `character` and `null`.

Booleans in Funcalc are represented as 1 and 0 for `true` and `false` respectively. For example, the `EQUAL` function compares its two arguments for equality and returns 1 or 0 as the result.

Funcalc does not differentiate between integers and floating-point numbers as SISAL does, instead everything is represented as a `double` in the implementation [3, section 2.8.2]. Consequently, Funcalc does not have the conversion functions for these numeric types and the boolean values 1 and 0 are `doubles` as well.

In SISAL, a string is represented as an array of characters: `array[character]`. Funcalc represents strings or text using the `TextValue` class and has no methods for manipulating text since users can call C# string manipulation functions using the `EXTERN` function. We will later see an example of this use case in section 3.8 where we need to count the words in a string.

SISAL has a `null` type that can be used to define enumerated types and base cases for recursive union types (see section 2.1.5) [1]. The `null` type contains only one value `nil` of type `null` which is used in definitions of SISAL's input/output language `Fibre` [1]. Funcalc has no `null` type and the closest analogue is probably a blank cell although there is no `ISBLANK` function as in Excel.

2.1.2 Array Types

SISAL supports n-dimensional arrays of a single element type such as `array[integer]` or `array[double_real]`. Arrays can be nested to create multi-dimensional arrays and powerful multi-dimensional indexing is supported.

As opposed to Excel, Funcalc supports first-class arrays and cells can contain arrays. The type of the contained elements is not given explicitly and is not limited to a single type. Unlike SISAL, Funcalc distinguishes between horizontal and vertical arrays as exemplified by some of of its intrinsic functions: `HSCAN`, `VSCAN`, `HARRAY` and `VARRAY` etc.

SISAL (and Excel) have more advanced functions for manipulating arrays compared to Funcalc, although Sestoft [3] has demonstrated that SDFs can be used to create Excel functions such as `GOALSEEK` and `VLOOKUP`.

SISAL supports sophisticated array indexing as shown in listing 1. We have created the `UPDATEARRAY` function for this purpose (see appendix A) because replace operations are used in the particle transport program in section 3.13. It can perform all SISAL index operations although it may require multiple calls since replace operations cannot be composed in the same update. As both SISAL and Funcalc arrays are immutable (or rather single-assignment in SISAL), the original array remains unmodified in both languages.

```
1  let
2      a := array[1: array[1: 1, 2, 3],
3                    array[1: 4, 5, 6],
4                    array[1: 7, 8, 9]]
5  in
6      a[1],                    % array[1: 1, 2, 3]
7      a[2, 3],                 % 6
8      a[2][3],                 % Same as previous
9      a[2: array[1: 0, 0, 0]], % Replaces the second row with a row of three zeroes
10     a[3, 3: 99],             % Replaces 9 with 99
11     a[1, 1: -1; 2, 2: -5],   % Negates the elements at (1, 1) and (2, 2)
12     a[3: 9, 8, 7]            % Replaces the bottom row with [9, 8, 7]
13 end let
```

Listing 1: Various ways of indexing and modifying arrays in SISAL.

Lastly, arrays can be concatenated in SISAL using the `||` operator. Arrays can be concatenated horizontally or vertically in Funcalc. Using `HCAT` and `VCAT` the resulting array is also flattened, but using `HARRAY` and `VARRAY`, no flattening occurs and the arguments are simply concatenated. If two arrays were passed as arguments, the result would be an array of arrays for

example.

2.1.3 Stream Types

SISAL streams are similar to arrays but prohibit random access and enforce sequential access. The literature on SISAL [1, 4] does not mention if streams are lazily evaluated as in other languages such as Haskell or Scala. They are only used in the Sieve of Eratosthenes program in section 3.7, an algorithm for generating prime numbers whose efficient functional implementation usually relies on lazy lists. Since streams are accessed differently than arrays they would be indistinguishable from arrays so are most likely lazy. Funcalc has no lazy evaluation except for using function values to delay evaluation.

2.1.4 Record Types

Record types are a collection of different data types much like `structs` in `C`, and can be nested to arbitrary depths. Funcalc has no such data structure, but we can emulate records using cell arrays and the `VLOOKUP` or `HLOOKUP` functions. Both of these were defined by Sestoft as SDFs [3, pp. 135–136]. Consider the "record" in sheet 1 that maps a person's name to their age.

	A	B
1	Alice	36
2	Bob	52
3	Eve	27

Sheet 1: A table of records for persons and their age in Funcalc.

The `A` column holds the *keys* of the record and column `B` holds the *values* for each key. The `VLOOKUP(x, arr, c)` function returns the column `c`, from the first row in array `arr` where the key is less than or equal to `x`. This poses a problem because SISAL records are one-to-one mappings that return values from *exact* key matches. Suppose we entered a new row in the record just before `Alice` called `Adam`. In this case, `Adam` is lexicographically less than `Alice` and `Adam`'s value would incorrectly be returned instead of `Alice`'s. Therefore, we need to define a new function `VLOOKUPX` that only returns exact matches (the `X` stands for the 'x' in exact) which uses a helper function called `MATCHROWX`. Excel provides these functions but they accept an additional parameter which controls whether the match is exact or approximate.

	A	B
1	`=DEFINE("matchrowx", B7, B2, B3, B4, B5)`	
2	`'key=`	`...`
3	`'i=`	`...`
4	`'rows=`	`...`
5	`'array=`	`...`
6	`'c=`	`...`
7	`'terminate?=`	`=B3>B4`
8	`'result=`	`=IF(B7, ERR("Element not found"), IF(EQUAL(INDEX(B5, 1, 1), B2), INDEX(B5, 1, B6), MATCHROWX(B2, B3 + 1, B4, SLICE(B5, 2, 1, ROWS(B5), COLUMNS(B5)), B6)))`

Sheet 2: SDF that finds the row which contains the key in its first column and returns the element in the same row and the `c`$^{\text{th}}$ column.

Let us break it down into pieces and examine them individually. The function checks whether the query key is the key of the current row. If it is, the value for that key is retrieved and returned, otherwise the next row is checked. The first `IF` statement checks the termination condition in cell `B7`: If our row counter `i` is bigger than the number of rows in the original array. If this is true, the key was not found and we raise an error. Otherwise, we move on to the next `IF` statement which checks if the item in the first row and column, i.e. the top-left corner of the array, is the key we are looking for. If it is, we return the value for that key which is assumed to be in the same row and in column `c` specified by cell `B6`, hence `INDEX(B5, 1, B6)`. If the key did not match, we instead invoke a recursive call to `MATCHROWX` incrementing the counter `i` and taking a slice of the array at position (2, 1). The slice indices are relatively to the array itself, so this slice will always cut off the first row (recall that Funcalc indices are 1-based). Thus the check of the second `IF` will actually inspect the key of the next row, or rather the first row of the new slice. Why are we passing a slice? Slices are *views* of their arrays so we save some space by not passing the full subarray at each recursive call.

We should also mention that a slightly more efficient implementation of the `MATCHROWX` function can be implemented using indices instead of passing slices to the recursive calls. With `MATCHROWX` defined, we can define `VLOOKUPX`.

SISAL records have unique keys as expected from a dictionary-like structure but the Funcalc functions we just defined do not stop you from defining a record with multiple instances of the same key. Funcalc cells are immutable so for record insertion and deletion, we would need to return a new array updated with the modifications. In general, emulating records in Funcalc

	A	B
1	=DEFINE("vlookupx", B5, B2, B3, B4)	
2	'key=	...
3	'array=	...
4	'c=	...
5	'result=	=MATCHROWX(B2, 1, ROWS(B3), B3, B4)

Sheet 3: A SDF similar to Sestoft's `VLOOKUP` function from [3, pp. 135–136], which returns exact key matches instead of keys that either match or are less than the query key.

seems more like a curiosity than a useful abstraction for end-users.

2.1.5 Union Types

SISAL union types are similar to ML's variants, Haskell's user-defined `data` types or F#'s discriminated unions, or unions from C. They can contain data of different types, but only one field is active at any given time and can be accessed. Like records, Funcalc has no native support for such a structure although it can also be emulated by having a data table as we did for the record and an accompanying index that marks the active field. It could also be mimicked using a 2-element array where the first element is the tag and the second is its value. In both cases, there is no support for type-checking or additional functionality, and we doubt whether they would be useful in a spreadsheet context.

As mentioned in section 2.1.1, `SISAL` supports recursive type definitions in `union` types. For instance the following code uses the `QTree` union inside its own definition [1, p. 23]:

```
1 % An enumerated type
2 type Quadrant = union[NE, SE, SW, NW: null];
3
4 % A recursive type definition
5 type QTree = union[Scalar: real;
6                    NonScalar: record[NE, SE, SW, NW: QTree]];
```

Listing 2: Defining enumerated and recursive types using SISAL's `null` type.

This also showcases how `null` can be used to declare enumerated types.

2.2 Let Expressions

Let expressions are found in many functional programming languages and SISAL is no exception. An example is shown in listing 3. The `let` construct constrains the scope of the variables declared in the first `let` part to the expression of the second `in` part. Funcalc has no `let` expression as there are no scoping rules for variables (e.g. one function is free to reference a cell used in another).

```
function AvgStddev(data: array[integer] returns integer, integer)
    let
        size := double_real(array_size(data));
        avg := for x in data
               returns value of sum double_real(x)
               end for / size;
        stddev := for x in data
                  returns value of sum exp(double_real(x) - avg, 2)
                  end for / size
    in
        avg, stddev
    end let
end function
```

Listing 3: Using a `let` expression to calculate the average and standard deviation of an array of integers. Notice how the for loop (product form) can be used in an expression.

2.3 Loops

SISAL provides two different loop constructs: The sequential *non-product* form and the parallel *product* form. Funcalc has no explicit loops but supports tail-recursive function calls which can substitute for looping. As of this writing, infinite recursive loops are not monitored and terminated, so they should be used with care. Theoretically all iterative loop forms in SISAL, barring parallelism, should be expressible in Funcalc using tail recursion.

2.3.1 Non-Product Form

This loop form is entirely sequential by design. SISAL will not attempt to parallelize them.

Using recursive function calls in Funcalc allows one to define any kind of SISAL non-product loop. The initial variable declarations are passed along

```
for initial
    I := 1
while I < 10649
    I := old I * 22
returns array of I
end for
```

Listing 4: Example of a non-product, sequential loop in SISAL. The result is the array `[1, 4: 1, 22, 484, 10648]`.

with the initial function call and the termination condition can either be hard-coded into the function or can use a function value given as argument. For counted loops, we can pass the initial value of the loop variable and the upper bound to recurse exactly a set number of user-defined times, or simply pass the upper bound and decrease it on each iteration until it reaches zero, if the loop variable itself is not important in the function.

2.3.2 Product Form

The independent statements of a product form loop can be executed in parallel and then aggregated in a `returns` statement. This is similar to the map-reduce or fold idiom found in functional programming, but with implicit support for parallel execution. An example loop is given in listing 5. In Funcalc, we can invoke multiple calls to the `MAP` functions then aggregate the results using `REDUCE`. More specifically, the SISAL language reference states that

> "*All computations that can be expressed by the product form can also be expressed by the non-product form. The converse is not true.*" [4]

Thus by extension, if Funcalc can express all non-product forms, then it must be able to express all product form loops. To allow parallel evaluation of such Funcalc constructs, they must be expressed in a fashion that permits efficient parallel computation, for example by splitting a product form loop, that does some computations on the elements of two arrays in parallel, into two `MAP` calls with possible overhead stemming from repeated computation if e.g an intermediate calculation is used in computing two otherwise independent results. The differences are illustrated in some of the example programs in section 3 such as the sequential and parallel versions of the approximation of π (section 3.4) and the retrieval of the index of the minimum element in

an array (section 3.6).

```
define main

function main(returns array[integer])
    A := array[1: 1, 2, 3, 4, 5]
    B := array[1: 6, 7, 8, 9, 10]

    for a, b in A dot B
    returns sum of a + b
    end for
end function
```

Listing 5: Computing the sum of the pairwise additions of two sequences. The result is 55. The additions can be done in parallel and the sum can be done as a reduction.

The `returns` clause can have a number of predefined *packaging* statements. In reality any aggregation statement that uses an associative operator can be used. They are discussed in detail in section 2.3.4.

2.3.3 Loop Indices

One can also iterate over the indices of elements in one or more arrays or ranges in SISAL.

```
% Array is [1: 4, 2, 7, 9, 1]
for a in array at i
returns sum of a * i
end for
% Result becomes 4 * 1 + 2 * 2 + 7 * 3 + 9 * 4 + 1 * 5 = 70
```

Listing 6: Enumerating the elements and their indices in SISAL. Notice that the lower bound of the array is set to 1 so we do not need to modify the index in the summation.

In Funcalc, the built-in function `TABULATE` provides looping over array indices. The `TABULATE` function would be trivial to run in parallel as the closure parameter is applied independently to each two-dimensional array index, and since arrays are immutable there is no fear of modifications to the original array during the tabulation. The closure must additionally be side-effect free to get deterministic results.

2.3.4 Result Packaging

SISAL's product form loop supports five ways of aggregating results:

- '`array of`' packages results into an array.
- '`stream of`' packages results into a stream.
- '`value of`' returns only the last value (filters apply).

We ignore `stream of` as Funcalc has no streams. Returning an array of values corresponds to a `MAP` operation in Funcalc, while just returning the last value of the operation can be done using a recursive function or a `MAP` operation followed by an `INDEX` operation that picks out the last value (although the latter is slightly less efficient).

There are also five associative reduction operations:

- `sum` sums the results.
- `product` multiplies the results.
- `greatest` returns the maximum value.
- `least` returns the minimum value.
- `catenate` concatenates arrays and streams.

For summation, Funcalc has the built-in function `SUM`. Multiplication of a set of values can be defined using `REDUCE` and a custom `PRODUCT` function that multiplies its two arguments and returns the result. For the maximum and minimum elements, we use `MAX` and `MIN` respectively. Finally for `catenate`, we can use `REDUCE` with an empty array as the initial value, and `HCAT` for concatenation, and `VCAT` vice versa for the vertical direction. The initial array will be prepended to the result, as it must be non-empty, and will have to be sliced from the result.

2.3.5 Loop Filtering

SISAL loops can contain filtering statements that remove certain results during iteration. For example, the following loop filters out all odd numbers in its iteration range.

```
1 for i in 1, 100
2 returns array of i when mod(i, 2) == 0
```

To create a filter function in Funcalc we can use a combination of `MAP` and `REDUCE` or a recursive SDF. The recursive approach is used in section 3.7.

2.3.6 Dot Products

Dot products only apply to product form loops and zips the arrays. Thus the following code returns `array[1: 5, 7, 9]` for `A := array[1: 1, 2, 3]` and `B := array[1: 4, 5, 6]`.

```
1 for a in A dot b in B
2 returns array of a + b
3 end for
```

Thanks to the generalised behaviour of `MAP`, zipping is already available.

2.3.7 Cross Products

Cross products allow for nested loops. The following SISAL code [1] produces a Hilbert matrix.

```
1 for i in 1, 2 cross j in 1, 2
2 returns array of 1.0 / real(i + j - 1)
3 end for
```

We could either use `TABULATE` to calculate the Hilbert matrix or a recursive function that keeps track of the indices of each loop which will also work for higher dimensional matrices.

2.4 Functions

Even though SISAL is a functional programming language, the literature does not indicate whether it is higher-order or not. None of the example programs from [1] make use of them and there are no types defined for functions, so we assume that SISAL is a first-order language. Other sources seem to claim that this is indeed the case [7]. The same source also claims that the proposal for SISAL 2.0 may include higher-order functions and other features commonly found in contemporary functional languages such as polymorphism and type inference. On the other hand, SISAL allows function definitions inside functions whereas Funcalc does not but it would likely be more confusing than useful. It is interesting that the resources

on SISAL [1, 4] do not emphasise higher-order functions and laziness (see section 2.1.3 on streams), both hallmarks of functional programming [8, 9] that promote modularity and reuse. In section 3, we shall show many examples of how higher-order functions can increase expressiveness and help solve a number of problems elegantly in Funcalc. SISAL does not support polymorphic or overloaded functions. In Funcalc, a function that takes three arguments and packages them into an array does not care what the type of the input values are, but that is the extent of the support. There are no overloaded functions in Funcalc as there cannot exist multiple functions with the same name.

Currently, Funcalc functions are limited to 10 input arguments. We assume SISAL has a similar constraint which is not specified in the language reference or tutorial or any other manual. To overcome this limitation, we can pass a subset of the arguments as an array and index each individual argument inside the function.

```
1 a, b, c = FunctionReturningThreeValues()
```

Unpacking function results in SDFs is currently disallowed in Funcalc. SISAL does not have any support for partial evaluation or function specialization, while Funcalc supports both. Finally, SISAL also supports recursion and possibly tail-recursion, being a functional language. Funcalc also supports recursion but does not guard against infinite recursive calls.

	A
1	={RETURN_THREE_RESULTS()}
2	={RETURN_THREE_RESULTS()}
3	={RETURN_THREE_RESULTS()}

Sheet 4: Unpacking three results returned by a function using an array formula into cells `A1`, `A2` and `A3`.

2.5 Errors

SISAL has a single error type which is parametrised on a given type depending on the situation. For example, `error[integer]` might be used for division by zero and `error[array[real]]` can signify that an expression in a `for` loop, producing an array, failed at some point. Funcalc returns error values directly in the affected cells. There is the `NA()` function which returns the special value `#NA`, signifying that a value was not available or yet to be given in a

closure. For function name errors, there is `#NAME?`. For anything else, the built-in function `ERR(msg)` can be used, which takes a single error message and outputs `#ERR: msg`. SISAL's error types can only communicate information about the error through its nested type. Accessing error values in an array in SISAL produces an error value of the corresponding element type, and inputs that are errors usually produce outputs that are errors too unless an erroneous part of an array is sliced away and the remaining array returned for example. Errors thus propagate through a SISAL program as needed, much like how the (NaN) error values in Funcalc were designed to propagate through expressions by clever exploitation of the IEEE 754-2008 floating-point standard [3, page 44, section 2.8.1].

2.6 Intrinsic Functions

SISAL provides a set of built-in functions, which we look at in turn and see if they have a Funcalc counterpart or can be expressed as a SDF. We ignore the functions that operate on streams.

SISAL functions with Funcalc equivalents are shown in the table below:

SISAL function	Funcalc Equivalent
abs	ABS
array_fill	CONSTARRAY
exp	x^y
mod	MOD
array_size	ROWS and COLUMNS
array_addl	HCAT and VCAT
array_addh	HCAT and VCAT
array_remh	SLICE
array_reml	SLICE
array_adjust	SLICE
floor	FLOOR(NA(), 1)
max	MAX
min	MIN
trunc	N/A

Table 1: The list of all predefined SISAL functions that have Funcalc equivalents. Note that for `array_reml` and `array_remh`, the low and high indices are also changed in SISAL, which is not the case in Funcalc.

In SISAL, the `trunc` function simply removes the non-integral part of the input, truncating towards zero [4] whereas the `FLOOR` function in Funcalc truncates towards $-\infty$ or $+\infty$ according to its second argument. An equivalent SDF can be defined by selecting the second argument to the `FLOOR`

function based on the sign of the input value. The `floor` function rounds towards negative infinity, so we need to give the Funcalc equivalent a positive number for its second argument in order to round the input in the same fashion. The `array_liml`, `array_limh` that set the low and high indices of an array, and `array_setl` cannot be expressed meaningfully in Funcalc as its arrays are always one-based and has no array limits.

3 Example Programs

To demonstrate the capabilities and limitations of Funcalc's expressiveness of SISAL programs, we have taken the example programs from the SISAL tutorial [1] and translated them to Funcalc using sheet-defined functions (SDFs).

Most of the programs are relatively simple and therefore we omit any explanation of their nature or intent. This reflects that this text is first and foremost a comparison between two programming paradigms, and not a primer on the subjects covered by the example programs. For instance, we do not explain how matrix multiplication works or what it is used for, but assume the reader already knows or can find out on his or her own.

We also omit error checking code in Funcalc to keep everything readable. The SISAL code has been stripped of its `Main` method where possible, along with explanatory comments and type aliases to keep things short. We have also beautified the code for better readability. Refer to the original SISAL tutorial for the unmodified code [1]. To improve readability, we reuse the top-left corner of the sheet for every function in the Funcalc function sheets, even those that are part of the same program. Note that this is not normally possible in Funcalc. Lastly, we abbreviate function arguments as three consecutive dots "..." if their definition is not important, which is usually the case for placeholder arguments in function definitions that exist so that the SDF can be evaluated in the sheet, or functions that take long arrays as arguments. SDFs are not necessarily as efficiently implemented as possible since we are more interested in Funcalc's ability to express SISAL programs than we are in performance. We follow the same convention for each program: The SISAL code is first presented followed by a discussion and step-wise presentation of the equivalent SDFs that make up the Funcalc translation of the SISAL program. Lastly, we will sometimes refer to appendix A for a list of oft used auxiliary functions.

3.1 Factorial Function

```
1  define Main
2
3  function Main(n: integer returns integer)
4      if (n <= 0) then 1 else n * Main(n - 1) end if
5  end function
```

Listing 7: The factorial function in SISAL.

The SDF is a straight-forward translation of the SISAL code.

	A	B
1	=DEFINE("fatorial", B3, B2)	
2	'n=	0
3	'out=	=IF(B2<=0, 1, B2*FACTORIAL(B2-1))

Sheet 5: Computing the factorial using recursion in Funcalc.

As an aside, it is straight-forward to design a tail-recursive version of the `FACTORIAL` function that uses an extra accumulator argument and is more efficient than the implementation given above. Such a function is given in subappendix A.2.

3.2 Matrix Multiplication

We first define the `SUMPRODUCT` function which is used for the matrix multiplication.

3.2.1 Sum of Products

The `SUMPRODUCT` is a well-known function from Excel which takes two arrays of similar shapes, multiplies elements pairwise and sums the products. It is similar to the mathematical dot product. For example, the `SUMPRODUCT` of the two arrays `[1, 2, 3]` and `[4, 5, 6]` is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$. See appendix A for the Funcalc implementation.

Defining the sum of products in SISAL is straight-forward by using a product form loop and the SISAL `dot` product to iterate over the two arrays in a pairwise fashion.

```
define SumProduct

function SumProduct(a: array[integer], b: array[integer] returns integer)
    for i, j in a dot b
    returns sum of i * j
    end for
end function
```

Listing 8: The well-known function from Excel in SISAL. It performs pairwise multiplication of the elements from both arrays, then sums the result.

In a functional programming context, the `zip` or `zip_with` functions come to mind since `SUMPRODUCT` is doing pairwise operations with a binary function which is followed by a reduction. Fortunately, Funcalc's `MAP` function has been generalised to operate on one or more arrays in a `zip_with` manner. In contrast, the ordinary `map` function in functional programming languages usually only operates on a single array or list, as showcased in the following `Scala` snippet:

```
    val a = List(1, 2, 3)
    a.map(_ + 1) // List(2, 3, 4)
```

The code defines a list of the values 1, 2 and 3, then maps that list to a new list using an anonymous function which adds 1 to each element of the original list. One would have to define `map2`, `map3` etc. for mapping multiple lists. Consequently, defining `SUMPRODUCT` in Funcalc is simple, and as an added bonus, it operates on an arbitrary number of horizontal or vertical arrays as well as two-dimensional arrays. However, because SDFs cannot be variadic, we have to define one that takes two arguments (see appendix A for its definition).

```
function Matmult(A, B: array[array[real]]; M, N, L: integer returns
 ↪  array[array[real]])
    for i in 1, M cross j in 1, L
        S := for k in 1, N
             returns value of sum A[i, k] * B[k, j]
             end for
    returns array of S
    end for
end function
```

Listing 9: Matrix multiplication in SISAL.

The `TABULATE` built-in function applies a function to each index `(r, c)` for

an array range given by a number of rows and columns. This is well suited to the matrix multiplication program. We start by defining a helper function, `MMULT_HELPER`.

	A	B
1	`=DEFINE("mmult_helper", B6, B2, B3, B4, B5)`	
2	`'array1=`	`...`
3	`'array2=`	`...`
4	`'r=`	`...`
5	`'c=`	`...`
6	`'out=`	`=SUMPRODUCT( SLICE(B2, B4, 1, B4, COLUMNS(B2)), SLICE(B3, B5, 1, B5, COLUMNS(B3)))`

Sheet 6: The helper function for computing the dot product of a single element in matrix multiplication.

The function takes the two input arrays (matrices) and a row and column index. It then uses the `SUMPRODUCT` function we defined in section 3.2.1 and slices off the appropriate row of the first matrix and column of the second matrix. The second vector is a column vector so we need to transpose it, because `SUMPRODUCT` internally uses `MAP` which expects its input arrays to have the same shape, but we do a single transposition in the main `MMULT` to avoid do transpositions for each element in the result matrix. The slices created in `MMULT_HELPER` are views of the original array and incur a lower cost of creation as opposed to creating an entire new array. We can now define `MMULT` binding the two arrays to the closure of `MMULT_HELPER`, transforming it into a function that needs only a row and column index, as expected by `TABULATE`.

	A	B
1	`=DEFINE("mmult", B4, B2, B3)`	
2	`'array1=`	`=...`
3	`'array2=`	`=...`
4	`'out=`	`=TABULATE(CLOSURE("MMULT_HELPER", B2, TRANSPOSE(B3), NA(), NA()), ROWS(B2), COLUMNS(B3))`

Sheet 7: Main matrix multiplication function.

3.3 Matrix Transposition

Matrix transposition is already provided by the built-in function `TRANSPOSE` in Funcalc, but for completeness, we provide it as an SDF `TRANSPOSE1` nonetheless. Instead of using two recursive functions to implement the

```
function Transpose(A: array[array[integer]] returns array[array[integer]])
    let
        N := array_size(A)
        M := array_size(A[1])
    in
        for i in 1, M cross j in 1, N
        returns array of A[j, i]
        end for
    end let
end function
```

Listing 10: Matrix transposition in SISAL.

nested for loop, we use `TABULATE` and a function for returning the transposed element for a given element position.

	A	B
1	`=DEFINE("transpose_element", B5, B2, B3, B4)`	
2	`'array=`	`=...`
3	`'r=`	`=...`
4	`'c=`	`=...`
5	`'result=`	`=INDEX(B2, B4, B3)`

Sheet 8: Getting the element in the resulting transposed matrix from the input matrix.

	A	B
1	`=DEFINE("transpose1", B3, B2)`	
2	`'array=`	`=...`
3	`'result=`	`=TABULATE(CLOSURE("transpose_element", B2, NA(), NA()), COLUMNS(B2), ROWS(B2))`

Sheet 9: Transposing a matrix using `TABULATE`. We call the function `TRANSPOSE1` to avoid name collision with the built-in `TRANSPOSE` function.

3.4 Calculating Pi

The SISAL tutorial [1] provides both sequential and parallel versions of algorithms for approximating π over a given number of iterations. Bear in mind that the two mathematical formulas used in the two functions are different.

3.4.1 Sequential Version

Funcalc has no notion of parallel functions or loops. Despite this limitation, we can implement both versions in Funcalc but without any parallelism. Note that Funcalc already provides a built-in function `PI` that returns `System.Math.PI` from `C#` represented as a `System.Double`.

```
define Main

function Main(Cycles: integer returns real)
    4.0 * for initial
              Approx := 1.0;
              Sign   := 1.0;
              Count  := 1
          while (Count < Cycles) repeat
              Sign   := -old Sign;
              Count  := old Count + 2
              Approx := old Approx + Sign / real(Count)
          returns
              value of Approx
          end for
end function
```

Listing 11: Sequential program for calculating the approximation of π.

Like most functions that use recursion, the current values of a given recursive function call must be passed along as function arguments. To hide this from the users of the `PISEQ` function (i.e. the sequential function for approximating π), a helper function is sometimes used. We will apply the same principle to the parallel Funcalc version.

	A	B
1	=DEFINE("piseq_helper", B9, B2, B3, B4, B5)	
2	'cycles=	...
3	'approx=	1
4	'sign=	1
5	'count=	1
6	'x=	B3+NEG(B4)/(B5+2)
7	'negsign=	NEG(B4)
8	'nextcount=	B5+2
9	'out=	=IF(B5<B2, PISEQ_HELPER(B2, B6, B7, B8), B6)

Sheet 10: The helper function for sequentially approximating π.

The recursive call in cell `B8` is in tail position making `PISEQ_HELPER` a tail-recursive function. Using `PISEQ_HELPER`, we can define `PISEQ`.

Note that the SDF compiler ensures that the intermediate cells `B6:B8` are

	A	B
1	=DEFINE("piseq", B3, B2)	
2	'cycles=	1000
3	'out=	=PISEQ_HELPER(B2, 1, 1, 1)*4

Sheet 11: SDF for sequentially approximating π.

only computed if they are needed by the output cell B9.

3.4.2 Parallel Version

```
1  define Main
2
3  function Main(Cycles: integer returns real)
4      (4.0/real(Cycles)) * for j in 1, Cycles
5                               x := (real(j) - 0.5) / real(Cycles)
6                           returns sum of 1.0 / (1.0 + x * x)
7                           end for
8  end function
```

Listing 12: Parallel approximation of π in SISAL.

Upon examining the function, we can see that it is essentially a summation of the values calculated by the loop and a single, final multiplication. We can thus use SUM and MAP with HSEQ (see appendix A) as an argument to MAP.

	A	B
1	=DEFINE("pipar_helper", B5, B2, B3)	
2	'j=	1
3	'cycles=	100
4	'x=	=(B2-0.5)/B3
5	'result=	=1/(1+B4*B4)

Sheet 12: Helper function for approximating π in parallel.

	A	B
1	=DEFINE("pipar", B5, B2, B3, B4)	
2	'cycles=	100
3	'range=	=HSEQ(1, B2, 1)
4	'fv=	=CLOSURE("PIPAR_HELPER", NA(), B2)
5	'result=	=SUM(MAP(B4, B3))*(4/B2)

Sheet 13: Funcalc function for estimating π in parallel.

3.5 Computing Statistics of An Array

```
function Stats(data: array[integer] returns double_real, double_real, double_real,
 ↪  double_real)
    let
        num := double_real(array_size(data));
        denom, maxv, minv, total := for x in data
                                        returns value of sum 1.0D0/x
                                        returns value of greatest x
                                        returns value of least x
                                        returns value of sum x
                                    end for;
        harm := num/denom;
        avg  := total/max(num, 1.0D0)
    in
        harm, avg, minv, maxv
    end let
end function
```

Listing 13: Calculating the minimal, maximal elements and the average and harmonic mean of an array.

All statistics, except for the harmonic mean have built-in functions: `AVERAGE` for the average, `MIN` for the smallest element and `MAX` for the largest. However, the harmonic mean is easily computed using `MAP` and `SUM` with a SDF that converts a number to its reciprocal.

	A	B
1	=DEFINE("reciprocal", B3, B2)	
2	'n=	=1
3	'result=	=1/B2

Sheet 14: Computing the reciprocal of a number.

	A	B
1	=DEFINE("stats", B3, B2)	
2	'array=	=HARRAY(2, 4.45, 10.9, 3.3, 2, 2, 1, 16)
3	'result=	=HARRAY(COLUMNS(B2)/SUM(MAP(CLOSURE("reciprocal"), B2)), AVERAGE(B2), MIN(B2), MAX(B2))

Sheet 15: Computing the harmonic mean, average, minimum and maximum values of an array in Funcalc.

Notice that we return a horizontal array in the `STATS` function. Since all the built-in functions in the `STATS` function work with two-dimensional arrays, the most general form of this function would return a horizontal or vertical array for the largest dimension of the input array. This could be done by

selecting the appropriate closure of either `HARRAY` or `VARRAY` and defaulting to one of them when the array is square. As described in section 2.3.4, we can use an array formula to unpack the results.

3.6 Index of the First Minimum Element

Like the approximation of π, the SISAL tutorial provides both sequential and parallel versions for finding the index of the first minimum element of an array.

3.6.1 Sequential Version

```
1  function ifmin(X: array[double_real] returns integer)
2      for initial
3          imin := 1; k := 2;
4      while (k <= array_size(X)) repeat
5          k := old k + 1;
6
7          imin := if (X[old k] < X[old imin])
8                      old k
9                  else
10                     old imin
11                 end if;
12     returns value of imin
13     end for
14 end function
```

Listing 14: Using a sequential non-product form loop for finding the index of the minimum element.

	A	B
1	=DEFINE("indexmin_helper", B7, B2, B3, B4)	
2	'array=	=...
3	'index=	=...
4	'min_index=	=...
5	'new_index=	=IF(INDEX(B2, 1, B3)<INDEX(B2, 1, B4), B3, B4)
6	'terminate?=	B3+1>COLUMNS(B2)
7	'result=	=IF(B6, B5, INDEXMIN_HELPER(B2, B3+1, B5))

Sheet 16: The `INDEXMIN_HELPER` function.

	A	B
1	=DEFINE("indexmin", B3, B2)	
2	'array=	=HARRAY(2, 4.45, 10.9, 3.3, 2, 2, 1, 16)
3	'result=	=INDEXMIN_HELPER(B2, 1, 1)

Sheet 17: The `INDEXMIN` function.

3.6.2 Parallel Version

The SISAL code seems less efficient than its sequential counterpart as it iterates through the array twice. First, we find the minimum element and then the index of this element by using a loop with a filter.

```
1  function ifmin(X: OneDim returns integer)
2      let
3          vmin := for Elm in X returns value of least Elm end for;
4      in
5          for Elm in X at I
6          returns value of least I when Elm = vmin
7          end for
8      end let
9  end function
```

Listing 15: Finding the index of first minimum element in an array in SISAL.

	A	B
1	=DEFINE("minindex", B4, B2, B3)	
2	'tuple1=	=HCAT(..., ...)
3	'tuple2=	=HCAT(..., ...)
4	'result=	=IF(INDEX(B3, 1, 2)<INDEX(B2, 1, 2), B3, B2)

Sheet 18: Comparing two tuples of elements and their indices.

We use `MIN` to find the minimum element `vmin` in the array and then iterate sequentially through the array, returning the index of the first element that has value `vmin`. Alternatively, we can `MAP` the elements of the array to arrays of their value and their index using `HSEQ` (subappendix A.5), then call `REDUCE` with a custom function that compares the values of two arrays and returns the index of the smaller one. Alternatively, we could define a `IMAP` or `IREDUCE`, with the `MIN` function, that keeps the indices of the elements around.

	A	B
1	=DEFINE("indexmin_par", B4, B2, B3)	
2	'array=	=...
3	'enumerated=	=MAP(CLOSURE("enum"), HSEQ(1, COLUMNS(B2), 1), B2)
4	'result=	=INDEX(REDUCE(CLOSURE("minindex"), HCAT(0, 1E+300), B3), 1, 1)

Sheet 19: The `INDEXMIN_PAR` function implemented using `MAP` and `REDUCE`.

We use the `ENUM` function to package in tuples, the indices generated by `HSEQ` and the array elements of B2. Note that we return zero when there is no minimal element in the case of an empty array. We pick a very large number for the second element in the initial array given to `REDUCE`, since we cannot access fields of types in Funcalc. More precisely, `System.Double.MaxValue` is a public, static field which is why we pass `null` to the final method in the following call.

```
System.Object.GetType(<double>).GetField("MaxValue").GetValue(null)
```

We cannot currently pass `null` values from Funcalc so we cannot retrieve the value. An alternative implementation of `INDEXMIN_PAR` can use `MIN` to get the minimal element and a recursive, auxiliary function to iterate the array by index and stop when the first occurrence of `vmin` is encountered and return its index.

3.7 Sieve of Eratosthenes

The Sieve of Eratosthenes is a classical algorithm for finding the prime numbers up to a given number.

```
1  global sqrt(a: double_real returns double_real)
2
3  function Filter(S: stream[integer]; M: integer returns stream[integer])
4      for I in S
5      returns stream of I unless mod(I, M) = 0
6      end for
7  end function
8
9  function Integers(Limit: integer returns stream[integer])
10     for initial
11         I := 3;
12     while I <= Limit repeat
13         I := old I + 2
14     returns stream of I
15     end for
16 end function
17
```

```
18 function main(Limit: integer returns stream[integer])
19     let
20         Maxt := integer(sqrt(double_real(Limit)))
21     in
22         for initial
23             S := Integers(Limit);
24             T := 2;
25         repeat
26             T := stream_first(old S);
27             S := if T <= Maxt then
28                      Filter(stream_rest(old S), T)
29                  else
30                      stream_rest(old S)
31                  end if
32         until stream_empty(S)
33         returns stream of T
34         end for
35     end let
36 end function
```

Listing 16: Sieve of Eratosthenes in SISAL.

The Sieve of Eratosthenes program is a good indication that SISAL streams are indeed lazy as this is the only program where streams are used in the entire SISAL tutorial and lazy streams are very useful to avoid allocating multiple, large arrays when computing the primes. This makes it impossible to implement the SISAL version of Sieve of Eratosthenes directly in Funcalc as we only have eager arrays.

We can use the `HSEQ` function (see subappendix A.5) to generate the initial array of numbers, referred to as `S` in the SISAL program. We also need to be able to filter numbers. For this we define a `FILTER` function as in sheet 20, and use an additional, initially empty, accumulator argument `acc` to make it tail-recursive[1].

Note how closely the definition of `FILTER` follows the structure of a typical functional implementation, where the head is examined and then conditionally appended to the accumulator and the filter function is applied to the tail of the array. An alternative, but perhaps slightly less efficient, approach of implementing `FILTER` uses `MAP` to map values that fail to satisfy the predicate to empty arrays, then use `REDUCE` to concatenate the mapped values with `HCAT`, which collapses the empty arrays.

Using `HSEQ` and `FILTER`, we can create a helper function for the main `PRIMES`

[1] This is a common optimisation technique in functional programming. Another technique for achieving tail-recursive is to use *continuations*, functions that contain the next computation.

	A	B
1	=DEFINE("filter", B5, B2, B3)	
2	'fv=	=CLOSURE("...")
3	'array=	=...
4	'acc=	=HARRAY()
5	'head=	=INDEX(B3, 1, 1)
6	'pred=	=APPLY(B2, B5)
7	'result=	=IF(COLUMNS(B3)=0, B4, FILTER(B2, SLICE(B3, 1, 2, ROWS(B3), COLUMNS(B3)), IF(B6, HCAT(B4, B5), B4)))

Sheet 20: Filtering an array in Funcalc using a tail-recursive function. Instead of passing slices around, we could just as easily pass indices around.

function which we dub _PRIMES. We use the same trick as with filtering by passing an additional accumulator argument to the _PRIMES function in order to make it tail-recursive. The implementation of the _MOD function has been omitted for brevity. It is like the built-in function MOD but with a negated condition to ensure that FILTER keeps the elements that are not divisible by its second input.

	A	B
1	=DEFINE("_primes", B5, B2, B3)	
2	'maxt=	=...
3	'integers=	=...
4	'acc=	=...
5	'head=	=INDEX(B3, 1, 1)
6	'tail=	=SLICE(B3, 1, 2, ROWS(B3), COLUMNS(B3))
7	'fv=	=CLOSURE("_mod", NA(), B5)
8	'result=	=IF(COLUMNS(B3)=0, B4, _PRIMES(B2, IF(B5<=B2, FILTER(B7, B5, HARRAY()), B6), HCAT(B4, B5)))

Sheet 21: The tail-recursive _PRIMES helper function in Funcalc.

We create a new closure of the _MOD function on each iteration that takes in the current head of the list as a second argument. This is passed to FILTER which then filters out any element that is divisible by the head.

Finally, we define PRIMES using _PRIMES passing in the appropriate arguments. We observe that 2 is the only even prime number while the rest are odd (ironically making 2 the oddest prime), so we can omit it from the integers array as in the SISAL program, and prepend it to the result of the call to _PRIMES. The initial array of integers can be created using =HSEQ(3, Limit, 2).

Note that this implementation uses only horizontal arrays. We can store the result in a single cell, but the result can only be used in a horizontal

	A	B
1	=DEFINE("primes", B4, B2)	
2	'limit=	=...
3	'maxt=	=FLOOR(SQRT(B2), 1)
4	'result=	=HCAT(2, _PRIMES(B3, HSEQ(3, B2, 2), HARRAY())

Sheet 22: Function for generating the primes.

array formula. If we attempt to use the result in a vertical one instead, we will only see the first argument and the rest will be filled with the error code `#NA`. We will encounter this many times during the translation of these programs, and provide suggestions for solving this issue in section 4.2.

3.8 Word Count

```
1  function is_char(c: character returns boolean)
2      if ((c = ' ') | (c = '\t') | (c = '\n')) then
3          false
4      else
5          true
6      end if
7  end function
8
9  function Main(text: array[character] returns integer)
10     for initial
11         count := 0; iword := 0; pointer := 1;
12     while (pointer <= array_size(text)) repeat
13         pointer := old pointer + 1;
14
15         count,
16         iword :=
17             if (is_char(text[old pointer]) & (old iword = 0)) then
18                 1, 1
19             elseif (is_char(text[old pointer]) & (old iword = 1)) then
20                 0, 1
21             else
22                 0, 0
23             end if;
24      returns value of sum count
25      end for
26 end function
```

Listing 17: Counting the number of words in a sentence in SISAL.

Funcalc has no functions for string manipulation. Thus this problem is not possible to express without the use of an external language. However, we can use the `EXTERN` function to call functions in `C#` such as its string utility functions. A word counting program in `C#` could be called with `EXTERN`

directly and wrapped in a SDF, or we could translate it to Funcalc using individual string methods with the help of `EXTERN`. We implement the latter approach to demonstrate more of the expressive power of Funcalc. We also assume the existence of functions `ISCHAR` and `INDEXAT` that use `EXTERN` to call the appropriate functions in `C#`. Both are available in subappendix A.6. Getting the string length involves calling its `Length` property. Sestoft [3] has already demonstrated this.

	A	B
1	=DEFINE("_wordcount", B7, B2, B3, B4, B5)	
2	'string=	="This sentence has five words"
3	'count=	=0
4	'iword=	=0
5	'pointer=	=1
6	'char=	=ISCHAR(INDEXAT(B2, B5))
7	'new_values=	=IF(AND(B6, B4)=0, HCAT(B3+1, 1), IF(AND(B6, B4)=1, HCAT(B3, 1), HCAT(B3, 0)))
8	'result=	=IF(B5>LEN(B2), B3, _WORDCOUNT(B2, INDEX(B7, 1, 1), INDEX(B7, 1, 2), B5+1))

Sheet 23: A helper function for recursively counting words in a string.

We can now easily define a `WORDCOUNT` function in the same manner as we have done before with other programs. The implementation in sheet 23 tries to follow the SISAL equivalent as closely as possible, but there are other more elegant ways to implement word counting. For example, we could have converted the string to an array and used `MAP` to map each character to one if it is located at the beginning of a word, zero otherwise, and then used `SUM` to count the words. Alternatively, we could just have used `EXTERN` to make the following call:

```
=EXTERN("System.Array.get_Length()I",
        EXTERN("System.String.Split([C)[T", string, null))
```

Unfortunately, we cannot currently pass a `C# null` value to external calls, so the latter will not work.

3.9 Batcher Sort

Batcher sort is one of many sorting algorithms that use sorting networks. It is an efficient, parallel algorithm with a worst-case running time of $O(log_2(n))$ compared with its sequential equivalent of $O(n \cdot log_2(n))$, due to independent compare-and-swap operations that can done in parallel.

```
function CeilingOfLog2(N: integer returns integer)
    for initial
        L := 0;
        TwoToTheL := 1
    while TwoToTheL < N repeat
        L := old L + 1;
        TwoToTheL := 2*old TwoToTheL
    returns value of L
    end for
end function

function FloorOfNOver2(N: integer returns integer)
    if N = 1 then 0 else N/2 end if
end function

function Isec(I, P: integer returns integer)
    if mod(I/P, 2) = 1 then P else 0 end if
end function

function Batcher(K: array[integer] returns array[integer])
    let
        N := array_size(K);
        T := CeilingOfLog2(N)
    in
        for initial
            P := exp(2, T);
            B := array_setl(K, 0)
        while P > 1 repeat
            P := FloorOfNOver2(old P);
            B := for initial
                     Q := exp(2, T);
                     R := 0;
                     D := P;
                     C := old B
                 repeat
                     C := for Elt in old C at I
                              NewElt := if (Isec(I, P) = old R) & (I + old D < N)
                               ↪   then
                                            min(Elt, old C[I + old D])
                                        elseif (Isec(I - old D, P) = old R) & (I >=
                                         ↪   old D) then
                                            max(Elt, old C[I - old D])
                                        else Elt
                                        end if;
                          returns array of NewElt
                          end for;
                     D := old Q - P;
                     Q := old Q / 2;
                     R := P
                 until Q < P
                 returns value of C
                 end for
         returns value of B
         end for
    end let
```

```
54 end function
```

Listing 18: The Bacther sort algorithm in SISAL.

To implement this program in Funcalc, we will work outwards from the inner part of the algorithm, factoring everything into functions that will be combined to form the complete sorting routine. Before continuing, note the call in line 27 which forces the lower bound of the SISAL array to start at zero. Bear this in mind for the Funcalc implementation as Funcalc arrays are one-based.

We do not need a separate function for `CeilingOfLog2`, as we can use the following equation instead: $\lceil log_2(2^{(log_{10}(x)/log_{10}(2))}) \rceil$. The logarithm function in base 2 is not natively available in Funcalc, but we can instead use `C#`'s mathematical library: `=EXTERN("System.Math.Log$(DD)D", 1024, 2)`.

The `FloorOfNOver2` function is already implemented in Funcalc by using ordinary division and the `FLOOR` function. We then implement the `Isec` function from line 16.

	A	B
1	`=DEFINE("isec", B4, B2, B3)`	
2	`'I=`	`=10`
3	`'P=`	`=2`
4	`'result=`	`=IF(FLOOR(MOD(B2/B3, 2), 1)=1, B3, 0)`

Sheet 24: The `Isec` function from the Batcher sort algorithm.

Getting into the meat of the sorting routine, we first implement a function for computing the new elements of the array `C` in the inner loop in lines 36 to 44. We have named it `BS_CMAP` as it is a function that effectively maps over `C`. The prefix stands for Batcher sort. Notice that we adjust the index before passing it to `ISEC` in both cases. For readability's sake, we have separated the first condition in the result and the value of `Elt`.

	A	B
1	=DEFINE("bs_cmap", B11, B2, B3, B4, B5, B6, B7, B8)	
2	'P=	=4
3	'N=	=15
4	'R=	=0
5	'D=	=5
6	'array=	=...
7	'r=	=1
8	'c=	=1
9	'cond1=	=AND(ISEC(B8-1, B2)=B4, B8+B5-1)
10	'Elt=	=INDEX(B6, B7, B8)
11	'result=	=IF(B9, MIN(B10, INDEX(B6, B7, B8+B5)), IF(AND(ISEC(B8-B5-1, B2)=B4, B8>B5), MAX(B10, INDEX(B6, B7, B8-B5)), B10))

Sheet 25: The mapping function for computing the new elements of `C` in Batcher sort.

We can now define a recursive function for the inner loop that calculates the `B` array in lines 30 and 50.

	A	B
1	=DEFINE("b_loop", B9, B2, B3, B4, B5, B6, B7)	
2	'P=	=4
3	'N=	=15
4	'Q=	=8
5	'R=	=0
6	'D=	=4
7	'array=	=...
8	'C=	=TABULATE(CLOSURE("BS_CMAP", B2, B3, B5, B6, B7, NA(), NA()), ROWS(B7), COLUMNS(B7))
9	'result=	=IF(B4<B2, B8, B_LOOP(B2, B3, FLOOR(B4/2, 1), B5, B4-B2, B8))

Sheet 26: The recursive `B_LOOP` function for Batcher sort.

Next is the familiar (recursive) helper function for wrapping the outermost loop in lines 25 and 52.

Finally, here is the `BATCHERSORT` function. We remark again that this function is tailored to work specifically with horizontal arrays. A call to an external sorting routine is probably more efficient.

	A	B
1	=DEFINE("batchersort_helper", B8, B2, B3, B4, B4, B6)	
2	'P=	=4
3	'T=	=15
4	'N=	=8
5	'array=	=...
6	'P_next=	=FLOOR(B2/2, 1)
7	'B=	=B_LOOP(B6, B4, 2^B3, 0, B6, B5)
8	'result=	=IF(B6>1, BATCHERSORT_HELPER(B6, B3, B4, B7), B7)

Sheet 27: Batcher sort recursive helper function.

	A	B
1	=DEFINE("batchersort", B6, B2)	
2	'array=	=...
3	'N=	=COLUMNS(B2)
4	'T=	=CEILING(LOG2(2^(LOG10(B3)/LOG10(2))), 1)
5	'P=	=2^B4
6	'B=	=BATCHERSORT_HELPER(B5, B4, B3, B2)

Sheet 28: The main Batcher sort function.

3.10 Gauss-Jordan Elimination Without Pivoting

We might consider using `ROWMAP` for Gauss elimination, but we encounter an issue because the function expects a function value that takes exactly as many arguments as there are columns in each row of the input. Consequently, we would have to supply a function for each possible number of columns we would want to support, which is neither scalable nor maintainable.

The Gauss elimination process expects a $N \times N$ matrix and a column vector of `N` rows. In the `Reduce` function in listing 19, each row of the `A` and `B` matrices are reduced in a way that is dependent only on the value of the pivot (lines 8 and 19), so it makes sense to split them into two different functions `GAUSS_REDUCEA` and `GAUSS_REDUCEB`. Because these operations use the current index along with other parameters, we can use `TABULATE`.

We then define the `GAUSS_HELPER` function which will be called from the main `GAUSS` function.

Further examining the SISAL code, we notice that for each row of the `A` matrix, each element in that row is reduced by an expression that is dependent on a conditional. Likewise, the elements of the single-row `B` vector are conditionally reduced for each n^2 times. This specific pattern works well

```
define Main

function Reduce(n, pivot: integer; A: array[array[double_real]]; B:
 ↪  array[double_real]
               returns array[array[double_real]], array[double_real])
   for i in 1, n
       row  := A[i];
       mult := A[i, pivot]/A[pivot, pivot];
       rA,
       rB := if i = pivot then
                 for j in 1, n
                 returns array of row[j]/A[pivot, pivot]
                 end for,
                 B[i] / A[pivot, pivot]
             else
                 for j in 1, n
                 returns array of row[j]-mult*A[pivot, j]
                 end for,
                 B[i]-mult*B[pivot]
             end if
   returns array of rA
           array of rB
   end for
end function

function Main(n: integer; Ain: array[array[double_real]]; Bin: array[double_real]
 ↪  returns array[double_real])
   for initial
       i    := 0;
       A, B := Ain, Bin;
   while i < n repeat
       i    := old i + 1;
       A, B := Reduce(n, i, old A, old B);
   returns value of B
   end for
end function
```

Listing 19: Gaussian elimination on matrices in SISAL.

	A	B
1	=DEFINE("gauss_reducea", B8, B2, B3, B4, B5)	
2	'pivot=	=1
3	'A=	=...
4	'r=	=1
5	'c=	=1
6	'elem=	=INDEX(B3, B4, B5)
7	'diag=	=INDEX(B3, B2, B2)
8	'reduced=	=IF(B4=B2, B6/B7, B6-INDEX(B3, B4, B2)/B7*INDEX(B3, B2, B5))

Sheet 29: Gauss reduction of the A matrix.

	A	B
1	=DEFINE("gauss_reduceb", B9, B2, B3, B4, B5, B6)	
2	'pivot=	=1
3	'A=	=...
4	'B=	=...
5	'r=	=1
6	'c=	=1
7	'elem=	=INDEX(B4, B5, B6)
8	'diag=	=INDEX(B3, B2, B2)
9	'reduced=	=IF(B5=B2, B7/B8, B7-INDEX(B3, B5, B2)/B8*INDEX(B4, B2, B6))

Sheet 30: Gauss reduction of the B matrix.

	A	B
1	=DEFINE("gauss_helper", B9, B2, B3, B4, B5, B6)	
2	'A=	=...
3	'B=	=...
4	'i=	=1
5	'n=	=ROWS(B2)
6	'rA=	=TABULATE(CLOSURE("GAUSS_REDUCEA", B4, B2, NA(), NA()), ROWS(B2), COLUMNS(B2))
7	'rB=	=TABULATE(CLOSURE("GAUSS_REDUCEB", B4, B2, B3, NA(), NA()), ROWS(B3), COLUMNS(B3))
8	'result=	=IF(B4<B5, GAUSS_HELPER(B6, B7, B4+1, B5), B7)

Sheet 31: The recursive Gauss reduction helper function.

	A	B
1	=DEFINE("gauss", B4, B2, B3)	
2	'A=	=...
3	'B=	=...
4	'result=	=GAUSS_HELPER(B2, B3, 1, ROWS(B2))

Sheet 32: The main Gauss reduction function.

with TABULATE as it did for matrix multiplication.

Using GAUSS_REDUCEA and GAUSS_REDUCEB along with TABULATE, we can define the GAUSS_HELPER function that recursively reduces A and B.

As per usual, we define a convenience function that wraps the recursive GAUSS_HELPER, and that expects a square matrix and a column vector. The issue of matrix shapes is obviated by the use of TABULATE that works with any shape.

3.11 Random Number Package

This package is a functional random number package i.e. the next pseudo-random number and the next seed are returned together to avoid stateful and non-pure functions. Funcalc provides the `RAND` function for generating a random number in the range of $[0, 1[$. Upon examining the SISAL code in listing 20, we see that all the functions in the package simply compute numbers. Thus the functions are readily translatable to Funcalc. We keep type aliases for this listing since they are heavily used and removing them would hamper readability.

```
1  type Four_Plex  = array[integer];
2  type Seed_Array = array[Four_Plex];
3  type Bit_Array  = array[integer];
4  type double     = double_real;
5
6  forward function ranf(Seed: Four_Plex returns double, Four_Plex)
7  forward function rans(N, Seed1: integer returns Seed_Array)
8  forward function ranf_a_to_k(a: Four_Plex; k: Bit_Array returns Four_Plex)
9  forward function ranf_even(n: integer returns integer)
10 forward function ranf_k(n: integer returns Four_Plex)
11 forward function ranf_k_binary(k: Four_Plex returns Bit_Array)
12 forward function ranf_mod_mult(a, b: Four_Plex returns Four_Plex)
13 forward function ranf_odd(n: integer returns integer)
14
15 function ranf(Seed: Four_Plex returns double, Four_Plex)
16     double_real(Seed[3]) / 4096.0d0 +
17     double_real(Seed[2]) / 16777216.0d0 +
18     double_real(Seed[1]) / 68719476736.0d0 +
19     double_real(Seed[0]) / 281474976710656.0d0,
20     ranf_mod_mult(array[0: 373, 3707, 1442, 647], Seed)
21 end function
22
23 function rans(N_In, Seed1: integer returns array[Four_Plex])
24     function N_Is_Odd(N: integer returns Boolean)
25         if mod(N, 2) = 1 then true else false end if
26     end function
27
28     for initial
29         N    := if N_is_Odd(N_In) then N_In else N_In + 1 end if;
30         i    := 1;
31         seed := if Seed1 = 0 then
32                     array[0: 3281, 4041, 595, 2376]
33                 else
34                     array[0: abs(Seed1), 0, 0, 0]
35                 end if;
36         a   := array[0: 373, 3707, 1442, 647];
37         a_k := if N > 1 then
38                    ranf_a_to_k(a, ranf_k_binary(ranf_k(N)))
39                else
40                    a
```

```
41              end if
42      while i < N repeat
43          i    := old i + 1;
44          seed := ranf_mod_mult(old seed, a_k)
45      returns array of seed
46      end for
47  end function
48
49  function ranf_a_to_k(a: Four_Plex; k: Bit_Array returns Four_Plex)
50      for initial
51          i   := 0;
52          a_i := a;
53          a_k := array[0: 1, 0, 0, 0]
54      while i < 46 repeat
55          i   := oldi + 1;
56          a_k := if k[i] = 0 then
57                     old a_k
58                 else
59                     ranf_mod_mult(old a_k, old a_i)
60                 end if;
61          a_i := ranf_mod_mult(old a_i, old a_i)
62      returns value of a_k
63      end for
64  end function
65
66  function ranf_even(n: integer returns integer)
67      if mod(n, 2) = 0 then 1 else 0 end if
68  end function
69
70  function ranf_k(n: integer returns Four_Plex)
71      let
72          nn := n + ranf_even( n );
73          q3 := 1024 / nn;
74          r3 := 1024 - (nn * q3);
75          q2 := (r3 * 4096) / nn;
76          r2 := (r3 * 4096) - (nn * q2);
77          q1 := (r2 * 4096) / nn;
78          r1 := (r2 * 4096) - (nn * q1);
79          q0 := (r1 * 4096) / nn
80      in
81          array [0: q0, q1, q2, q3]
82      end let
83  end function
84
85  function ranf_k_binary(k: Four_Plex returns Bit_Array)
86      for i in 0, 3
87      returns value of catenate
88          for initial
89              j   := 1;
90              x   := k[i] / 2;
91              bit := ranf_odd(k[i])
92          while j < 12 repeat
93              j   := old j + 1;
94              x   := old x / 2;
95              bit := ranf_odd(old x)
96          returns array of bit
```

```
97          end for
98      end for
99  end function
100
101 function ranf_mod_mult(a, b: Four_Plex returns Four_Plex)
102     let
103         j0 := a[0] * b[0];
104         j1 := a[0] * b[1] + a[1] * b[0];
105         j2 := a[0] * b[2] + a[1] * b[1] + a[2] * b[0];
106         j3 := a[0] * b[3] + a[1] * b[2] + a[2] * b[1] + a[3] * b[0];
107         k0 := j0;
108         k1 := j1 + k0 / 4096;
109         k2 := j2 + k1 / 4096;
110         k3 := j3 + k2 / 4096
111     in
112       array [0: mod(k0, 4096), mod(k1, 4096),
113                  mod(k2, 4096), mod(k3, 4096)]
114     end let
115 end function
116
117 function ranf_odd(n: integer returns integer)
118     if mod(n, 2) = 0 then 0 else 1 end if
119 end function
```

Listing 20: A random number package written in SISAL.

In the Funcalc implmentation, we have chosen to leave out the `ranf_odd` and `ranf_even` functions and directly use `MOD(n, 2)` to determine parity. We also leave out the embedded `N_Is_Odd` function as truth values in Funcalc are 1 and 0 which is already returned by the `MOD` built-in.

	A	B
1	=DEFINE("ranf_mod_mult", B12, B2, B3)	
2	'a=	=...
3	'b=	=...
4	'j0=	=INDEX(B2, 1, 1)*INDEX(B3, 1, 1)
5	'j1=	=INDEX(B2, 1, 1)*INDEX(B3, 1, 2)+INDEX(B2, 1, 2)*INDEX(B3, 1, 1)
6	'j2=	=INDEX(B2, 1, 1)*INDEX(B3, 1, 3)+INDEX(B2, 1, 2)*INDEX(B3, 1, 2)+INDEX(B2, 1, 3)*INDEX(B3, 1, 1)
7	'j3=	=INDEX(B2, 1, 1)*INDEX(B3, 1, 4)+INDEX(B2, 1, 2)*INDEX(B3, 1, 3)+INDEX(B2, 1, 3)*INDEX(B3, 1, 2)+INDEX(B2, 1, 4)*INDEX(B3, 1, 1)
8	'k0=	=B4
9	'k1=	=B5+B8/4096
10	'k2=	=B6+B9/4096
11	'k3=	=B7+B10/4096
12	'result=	=HARRAY(MOD(B8, 4096), MOD(B9, 4096), MOD(B10, 4096), MOD(B11, 4096))

Sheet 33: The `RANF_MOD_MULT` function from the random package.

	A	B
1	=DEFINE("ranf", B5, B2)	
2	'seed=	=...
3	'temp=	=INDEX(B2, 1, 4)/4096+INDEX(B2, 1, 3)/4096^2 +INDEX(B2, 1, 2)/4096^3+INDEX(B2, 1, 1)/4096^4
4	'new_seed=	=RANF_MOD_MULT(HARRAY(373, 3707, 1442, 647), B2)
5	'result=	=HARRAY(B3, B4)

Sheet 34: The RANF function from the random package. We have replaced the number constants from the SISAL code with their powers of 4096.

	A	B
1	=DEFINE("ranf_k", B12, B2)	
2	'n=	=10
3	'nn=	=B2+NOT(MOD(B2, 2))
4	'q3=	=1024/B3
5	'r3=	=1024-B3*B4
6	'q2=	=B5*4096/B3
7	'r2=	=B5*4096-B3*B6
8	'q1=	=B7*4096/B3
9	'r1=	=B7*4096-B3*B8
10	'q0=	=B9*4096/B3
11	'result=	=HARRAY(B10, B8, B6, B4)

Sheet 35: The RANF_K function from the random package.

	A	B
1	=DEFINE("bitarray_helper", B6, B2, B3, B4, B5)	
2	'n=	=3
3	'start=	=1
4	'nbits=	=12
5	'acc=	=HARRAY()
6	'result=	=IF(B3<=B4, BITARRAY_HELPER(FLOOR(B2/2, 1), B3+1, B4, HCAT(MOD(B2, 2), B5)), B5)

Sheet 36: The BITARRAY_HELPER function for recursively generating a bitarray.

	A	B
1	=DEFINE("bitarray", B4, B2)	
2	'n=	=3
3	'nbits=	=12
4	'result=	=BITARRAY_HELPER(B2, 1, B3, HARRAY())

Sheet 37: The BITARRAY function for generating a bitarray.

	A	B
1	=DEFINE("ranf_k_binary", B4, B2)	
2	'a=	=...
3	'bitarrays=	=MAP(CLOSURE("BITARRAY", NA(), 12), B2)
4	'bitarray=	=REDUCE(CLOSURE("HCAT2"), HARRAY(), B3)

Sheet 38: The RANF_K_BINARY function from the random package.

In RANF_K_BINARY, we use the HCAT2 which is simply a wrapper around HCAT to circumvent the current limitation where built-in functions cannot be bound to closures at the time of writing. This limitation has since been overcome.

	A	B
1	=DEFINE("ranf_a_to_k", B4, B2, B3)	
2	'a=	=...
3	'k=	=...
4	'result=	=RANF_A_TO_K_HELPER(1, B3, B2, HARRAY(1, 0, 0, 0))

Sheet 39: The RANF_A_TO_K function in the random package. The input k is a bitarray.

	A	B
1	=DEFINE("ranf_a_to_k_helper", B6, B2, B3, B4, B5)	
2	'i=	=1
3	'k=	=...
4	'a_i=	=...
5	'a_k=	=...
6	'result=	=IF(B2<=46, RANF_A_TO_K_HELPER(B2+1, B3, RANF_MOD_MULT(B4, B4), IF(INDEX(INDEX(B3, 1, B2)=0, B5, RANF_MOD_MULT(B5, B4))), B5))

Sheet 40: The RANF_A_TO_K_HELPER function in the random package.

	A	B
1	=DEFINE("rans_helper", B8, B2, B3, B4, B5, B6)	
2	'i=	=1
3	'n=	=2
4	'seed=	=...
5	'a_k=	=...
6	'acc=	=HARRAY()
7	'temp?=	=RANF_MOD_MULT(B4, B5)
8	'result=	=IF(B2<=B3, RANS_HELPER(B2+1, B3, B7, B5, HCAT(B6, HARRAY(B7))), SLICE(B6, 1, 2, ROWS(B6), COLUMNS(B6)))

Sheet 41: The RANS_HELPER function from the random package.

	A	B
1	=DEFINE("rans", B8, B2, B3)	
2	'n=	=2
3	'seed1=	=...
4	'm=	=B2+NOT(MOD(B2, 2))
5	'seed=	=IF(B3=0, HARRAY(3281, 4041, 595, 2376))
6	'a=	=HARRAY(373, 3707, 1442, 647)
7	'a_k=	=IF(B4>1, RANF_A_TO_K(B6, RANF_K_BINARY(RANF_K(B4))), B6)
8	'result=	=RANS_HELPER(1, B4, B5, HARRAY(HARRAY()), B7)

Sheet 42: The RANS function in the random package.

There was only one slight problem with translating the SISAL code. Notice that the `rans` function returns an `array[Four_Plex]` i.e. an array of arrays of integers since `Four_Plex` is a type alias for `array[integer]`. If we use `HCAT` to recursively construct the result array, we end up flattening and concatening the arrays, yielding a one-dimensional array, because `HCAT` unpacks its second argument concatenates it to the first: `=HCAT(HARRAY(1, 2), HARRAY(3, 4))` is `HARRAY(1, 2, 3, 4)`. On the other hand, `HARRAY` concatenates arrays without any unpacking, so if we have two arrays in an array, e.g. `HARRAY(HARRAY(1, 2), HARRAY(3, 4))`, and we concatenate a third array, we end up with an array of two elements: An array of arrays in the first element, and a single array in the second argument. Neither outcome is what we need to return the correct array. Instead we can use `HCAT` and `HARRAY` together to get the behaviour we are looking for. We pass an empty, horizontal array of arrays to the accumulator of the `RANS_HELPER` function (line 44), and then use `HCAT` with the accumulator array and the next array to be concatenated packaged inside a `HARRAY`. Due to the described behaviour of these functions we get the intended behaviour: `=HCAT(HARRAY(1, 2, 3), HARRAY(HARRAY(4, 5, 6)))` becomes

`=HARRAY(HARRAY(1, 2, 3), HARRAY(4, 5, 6))`. If we really wanted to, we could define a separate SDF for appending arrays to arrays of arrays as in sheet 43.

	A	B
1	`=DEFINE("happend", B4, B2, B3)`	
2	`'2d_array=`	`=HARRAY(HARRAY(1, 2), HARRAY(3, 4))`
3	`'1d_array=`	`=HARRAY(5, 6)`
4	`'result=`	`=HCAT(HARRAY(B2), HARRAY(B3))`

Sheet 43: Using `HCAT` and `HARRAY` to append one-dimensional arrays to two-dimensional arrays.

3.12 Conway's Game of Life

In Conway's Game of Life, a number of cells are located on a grid where a cell is denoted by 1, empty spaces by 0. Cells are updated according to a set of predefined rules based on the number of neighbours a cell has. We state the three rules as defined in the code of the SISAL program below.

- If a cell has more than five neighbors, then it should be a 0.
- If an empty space does not have exactly three neighbors, then it should be a 1.
- Otherwise, the position in the grid remains unchanged.

Conway's Game of Life has an incredibly elegant solution in Funcalc using immutable arrays, powerful built-in functions and array slicing. The original SISAL code included functions for generating a random grid of cells using the random number generation package from section 3.11, but we have omitted it here and instead focused on the functions that compute each iteration. Again, the `TABULATE` function suits our needs as we can update each cell by looking at the input array at each cell position.

First, we need a function to count the number of neighbours of a given cell at some position the in grid.

Sheet 44 is quite a simple and elegant solution. We slice off the appropriate subarray for the cell's neighbours at the given row and column index, then calculate their sum. Finally, we subtract the value in the current cell to avoid counting the cell itself towards its number of neighbours. Since cells

```
function Compute(G: Grid; I, J: integer; returns integer)
    let
        Total := G[I-1,J-1] + G[I-1,J] + G[I-1,J+1] +
                 G[I,J-1]         +       G[I,J+1]   +
                 G[I+1,J-1] + G[I+1,J] + G[I+1,J+1];
    in
        if     (G[I, J] = 1 & Total > 5)  then 0
        elseif (G[I, J] = 0 & Total ~= 3) then 1
        else G[I, J] end if
    end let
end function

function DoWork(G: Grid; Rows, Columns: integer returns Grid)
    let
        First := for i in 0, Columns + 1 returns array of G[0, i] end for;
        Last  := for i in 0, Columns + 1
                 returns array of G[Rows + 1, i]
                 end for;
        Core  := for I in 1, Rows
                     Mid := for J in 1, Columns
                            returns array of Compute(G, I, J)
                            end for;
                     Row := array_addl(Mid, G[I,0]);
                 returns array of array_addh(Row, G[I, Columns + 1])
                 end for;
    in
        array_addl(array_addh(Core,Last),First)
    end let
end function
```

Listing 21: Updating the cells in Conway's Game of Life in SISAL.

	A	B
1	=DEFINE("neighbours", B5, B2, B3, B4)	
2	'array=	=...
3	'r=	=1
4	'c=	=1
5	'count=	=SUM(SLICE(B2, B3-1, B4-1, B3+1, B4+1))-INDEX(B2, B3, B4)

Sheet 44: Calculating the number of neighbours for a cell in Conway's Game of Life.

are either 0 or 1, subtraction is a no-op for empty cells. Next, we define the `UPDATE` function for updating a cell. In the corresponding SISAL program, the empty border cells are added around the result of each update and returned. While this is certainly possible in Funcalc using `HCAT`, `VCAT` and some range generation functions, we opt for a more simplistic solution that

sets a cell to zero if it appears as part of the border.

	A	B
1	=DEFINE("update", B7, B2, B3, B4)	
2	'array=	=...
3	'r=	=1
4	'c=	=1
5	'border?=	=OR(B3=1, B4=1, B3=ROWS(B2), B4=COLUMNS(B2))
6	'neighbours=	=NEIGHBOURS(B2, B3, B4)
7	'old_cell=	=INDEX(B2, B3, B4)
8	'new_cell=	=IF(B5, 0, IF(AND(B7=1, B6>5), 0, IF(AND(B7=0, B6<>3), 1, B7)))

Sheet 45: Updating a cell in Conway's Game of Life.

Since this program involves bounded recursion, the astute reader may already have guessed that we need a helper function to provide a nice interface to users. This function is defined in sheet 45 and the `CONWAY` function is defined in sheet 47.

	A	B
1	=DEFINE("conway_helper", B6, B2, B3, B4)	
2	'i=	=1
3	'iterations=	=4
4	'array=	=...
5	'updater=	=CLOSURE("update", B4, NA(), NA())
6	'result=	=IF(B2<B3, CONWAY_HELPER(B2+1, B3, TABULATE(B5, ROWS(B4), COLUMNS(B4))), B4)

Sheet 46: The recursive helper function for Conway's Game of Life.

	A	B
1	=DEFINE("conway", B4, B2, B3)	
2	'iterations=	=4
3	'array=	=...
4	'result=	=CONWAY_HELPER(1, B2, B3)

Sheet 47: The main function for generating a number of iterations in Conway's Game of Life.

3.13 Particle Transport

```
1  function reflect(x, pivot, xmax, delta: double_real returns double_real)
2      let
3          frac := x - double_real(trunc(x / xmax)) * xmax
4      in
5          if frac = 0.0d0 then pivot - delta else pivot - frac end if
```

```
    end let
end function

function move(np, xcell, ycell: integer;
              dt, q, mass, xmax: double_real;
              ymax, xpcell, ypcell: double_real;
              xin, yin, vxin, vyin: array[double_real];
              returns array[double_real],
                      array[double_real],
                      array[double_real],
                      array[double_real])
    let
        cell, wght :=
            for i in 1, np
                r_row := yin[i] / ypcell + 1.0d0;
                r_col := xin[i] / xpcell + 1.0d0;
                row   := trunc(r_row);
                col   := trunc(r_col);
                lft   := r_col - double_real(col);
                rht   := 1.0d0 - lft;
                bot   := r_row - double_real(row);
                top   := 1.0d0 - bot;
                cell  := array[1: row, col];
                wght  := array[1: lft, top, rht, bot]
            returns array of cell
                    array of wght
            end for;

        grids := for i in 1, 10
                 returns array of
                     for initial
                         k   := (i - 1) * np / 10;
                         ep  := k + np / 10;
                         row := array_fill(1, xcell + 1, 0.0d0);
                         rho := array_fill(1, ycell + 1, row)
                     while k < ep repeat
                         k   := old k + 1;
                         r   := cell[k, 1];
                         c   := cell[k, 2];
                         qsw := q * wght[k, 2] * wght[k, 3];
                         qse := q * wght[k, 1] * wght[k, 2];
                         qnw := q * wght[k, 3] * wght[k, 4];
                         qne := q * wght[k, 1] * wght[k, 4];
                         rho := old rho[r, c:         old rho[r, c] + qsw;
                                        r, c + 1:     old rho[r, c + 1] + qse;
                                        r + 1, c:     old rho[r + 1, c] + qnw;
                                        r + 1, c + 1: old rho[r + 1, c + 1] + qne]
                     returns value of rho
                     end for
                 end for;

        rho := for i in 1, ycell + 1 cross j in 1, xcell + 1
               returns array of
                   for k in 1, 10 returns value of sum grids[k, i, j] end for
               end for;

```

```
62          esp := for initial
63                     i   := 0;
64                     pi  := 3.1415926d0;
65                     dx2 := xpcell * xpcell;
66                     esp := for y in rho at i, j
67                            returns array of pi * y * dx2
68                            end for
69                 while i < 10 repeat
70                     i   := old i + 1;
71                     esp := for y in old esp at i, j
72                                w := if j = 1 then 0.0d0
73                                     else old esp[i, j - 1] end if;
74                                n := if i = 1 then 0.0d0
75                                     else old esp[i - 1, j] end if;
76                                e := if j = xcell + 1 then 0.0d0
77                                     else old esp[i, j + 1] end if;
78                                s := if i = ycell + 1 then 0.0d0
79                                     else old esp[i + 1, j] end if;
80                                z := pi * y * dx2 + (w + n + e + s)/4.0d0
81                            returns array of z
82                            end for
83                 returns value of esp
84                 end for;
85
86          ax, ay := for k in 1, np
87                        i   := cell[k, 1];
88                        j   := cell[k, 2];
89                        bot := (esp[i, j] - esp[i, j + 1]) / xpcell;
90                        top := (esp[i + 1, j] - esp[i + 1, j + 1]) / xpcell;
91                        lft := (esp[i, j] - esp[i + 1, j]) / ypcell;
92                        rgt := (esp[i, j + 1] - esp[i + 1, j + 1]) / ypcell;
93                        ex  := top * wght[k, 4] + bot * wght[k, 2];
94                        ey  := lft * wght[k, 3] + rgt * wght[k, 1]
95                    returns array of ex * q / mass
96                            array of ey * q / mass
97                    end for;
98
99          vx1, vy1 := for i in 1, np
100                       returns array of vxin[i] + ax[i] * dt
101                               array of vyin[i] + ay[i] * dt
102                       end for;
103
104         x1, y1 := for i in 1, np
105                   returns array of xin[i] + vx1[i] * dt
106                           array of yin[i] + vy1[i] * dt
107                   end for;
108
109         x, vx, y, vy :=
110             for i in 1, np
111                 delta := 0.0000000001d0;
112                 x, vx := if x1[i] < 0.0d0 then
113                              reflect(x1[i], 0.0d0, xmax, -delta), -vx1[i]
114                          elseif x1[i] = 0.0d0 then
115                              x1[i] + delta, -vx1[i]
116                          elseif x1[i] > xmax then
117                              reflect(x1[i], xmax, xmax, delta), -vx1[i]
```

```
118                         elseif x1[i] = xmax then
119                             x1[i] - delta, -vx1[i]
120                         else
121                             x1[i], vx1[i]
122                         end if;
123                 y, vy := if y1[i] < 0.0d0 then
124                              reflect(y1[i], 0.0d0, ymax, -delta), -vy1[i]
125                          elseif y1[i] = 0.0d0 then
126                             y1[i] + delta, -vy1[i]
127                          elseif y1[i] > ymax then
128                             reflect(y1[i], ymax, ymax, delta), -vy1[i]
129                          elseif y1[i] = ymax then
130                             y1[i] - delta, -vy1[i]
131                          else
132                             y1[i], vy1[i]
133                          end if
134             returns array of x
135                     array of vx
136                     array of y
137                     array of vy
138             end for
139     in
140         x, y, vx, vy
141     end let
142 end function
```

Listing 22: The particle transport program for simulating the movements of particles in a cell.

The particle transport function may seem large and complicated, but on closer examination, we discover that it is essentially a single big function `move` that only uses arithmetic and loops, and should therefore be readily expressible in Funcalc. The only hurdle is the array update in lines 36 to 54 that updates a part of the `rho` array. There is no intrinsic function in Funcalc that provides this functionality, but we can define our own function `UPDATEARRAY` that provides us with this functionality. Refer to subappendix A.4 for details.

	A	B
1	=DEFINE("reflect", B7, B2, B3, B4, B5)	
2	'x=	=0.745425
3	'pivot=	=0
4	'xmax=	=1
5	'delta=	=0.1
6	'frac=	=B2-FLOOR(B2/B4, 1)*B4
7	'result=	=IF(B6=0, B3-B5, B3-B6)

Sheet 48: The `REFLECT` function from the SISAL particle transport program in Funcalc.

We now define functions for calculating the `cell` and `wght` variables in lines 18 to 32.

	A	B
1	`=DEFINE("cell", B6, B2, B3, B4, B5)`	
2	`'xin_i=`	`=0.5`
3	`'yin_i=`	`=0`
4	`'xpcell=`	`=1/4`
5	`'ypcell=`	`=1/4`
6	`'result=`	`=HCAT(FLOOR(B3/B5+1, 1), FLOOR(B2/B4+1, 1))`

Sheet 49: The `CELL` function from the SISAL particle transport program in Funcalc which calculates the new cell vector.

	A	B
1	`=DEFINE("wght", B14, B2, B3, B4, B5)`	
2	`'xin_i=`	`=0.987245`
3	`'yin_i=`	`=0.24854`
4	`'xpcell=`	`=1/4`
5	`'ypcell=`	`=1/4`
6	`'r_row=`	`=B3/B5+1`
7	`'r_col=`	`=B2/B4+1`
8	`'row=`	`=FLOOR(B6, 1)`
9	`'col=`	`=FLOOR(B7, 1)`
10	`'lft=`	`=B7-B9`
11	`'rht=`	`=1-B10`
12	`'bot=`	`=B6-B8`
13	`'top=`	`=1-B12`
14	`'result=`	`=HCAT(B10, B13, B11, B12)`

Sheet 50: The `WGHT` function which calculates the new weight vector.

Next, we define a function for computing the `grid` variable in lines 34 to 55. The loop runs for exactly ten iterations so instead of defining a recursive function as we would perhaps normally do, we instead use `MAP` in conjunction with `HSEQ` (see subappendix A.5 in appendix A), but first we need a function for the inner loop that computes the `rho` array.

	A	B
1	=DEFINE("rho", B19, B2, B3, B4, B5, B6)	
2	'k=	=9
3	'ep=	=10
4	'rho=	=CONSTARRAY(...)
5	'cell=	=HARRAY(...)
6	'wght=	=HARRAY(...)
7	'r=	=INDEX(INDEX(B5, 1, B2+1), 1, 1)
8	'c=	=INDEX(INDEX(B5, 1, B2+1), 1, 2)
9	'qsw=	=B18*INDEX(INDEX(B6, 1, B2+1), 1, 2)*INDEX(INDEX(B6, 1, B2+1), 1, 3)
10	'qse=	=B18*INDEX(INDEX(B6, 1, B2+1), 1, 1)*INDEX(INDEX(B6, 1, B2+1), 1, 2)
11	'qnw=	=B18*INDEX(INDEX(B6, 1, B2+1), 1, 3)*INDEX(INDEX(B6, 1, B2+1), 1, 4)
12	'qne=	=B18*INDEX(INDEX(B6, 1, B2+1), 1, 1)*INDEX(INDEX(B6, 1, B2+1), 1, 4)
13	'qsw1=	=INDEX(B4, B7, B8)+B9
14	'qse1=	=INDEX(B4, B7, B8+1)+B10
15	'qnw1=	=INDEX(B4, B7+1, B8)+B11
16	'qne1=	=INDEX(B4, B7+1, B8+1)+B12
17	'next_rho=	=UPDATEARRAY(B4, VCAT(HCAT(B13, B14), HCAT(B15, B16)), B7, B8)
18	'q=	=1
19	'result=	=IF(B2+1<B3, RHO(B2+1, B3, B17, B5, B6), B17)

Sheet 51: The RHO function which computes the rho variable.

	A	B
1	=DEFINE("grids", B11, B2, B3, B4, B5, B6)	
2	'np=	=10
3	'xcell=	=4
4	'ycell=	=4
5	'cell=	=HARRAY(...)
6	'wght=	=HARRAY(...)
7	'rho=	=CONSTARRAY(0, B4+1, B3+1)
8	'ks=	=HCAT((1-1)*B2/10, (2-1)*B2/10, ...)
9	'eps=	=HCAT(INDEX(B8, 1, 1)+B2/10, INDEX(B8, 1, 2)+B2/10, ...)
10	'fv=	=CLOSURE("rho", NA(), NA(), B7, B5, B6)
11	'result=	=MAP(B10, B8, B9)

Sheet 52: The GRIDS function for computing the grids variable, implemented using MAP.

The next variable in line 57 is also called rho but is declared in a different scope. It can be implemented elegantly: =MAP(CLOSURE("SUM"), GRIDS(...)). We only need to wrap the SUM function in a SDF, so that we can bind it to a closure.

	A	B
1	=DEFINE("esp", B9, B2, B3, B4, B5, B6)	
2	'i=	=1
3	'dx2=	=0.25^2
4	'esp=	=CONSTARRAY(...)
5	'xcell=	=4
6	'ycell=	=4
7	'fv=	=CLOSURE("newesp", B4, B3, B5, B6, NA(), NA())
8	'new_esp=	=TABULATE(B7, ROWS(B4), COLUMNS(B4))
9	'result=	=IF(B2<10, ESP(B2+1, B3, B8, B5, B6), B8)

Sheet 53: The ESP function for computing the esp variable.

	A	B
1	=DEFINE("newesp", B12, B2, B3, B4, B5, B6, B7)	
2	'esp=	=CONSTARRAY(...)
3	'xcell=	=4
4	'ycell=	=4
5	'dx2=	=0.25^2
6	'r=	=5
7	'c=	=5
8	'w=	=IF(B7=1, 0, INDEX(B2, B6, B7-1))
9	'n=	=IF(B6=1, 0, INDEX(B2, B6-1, B7))
10	'e=	=IF(B7=B3+1, 0, INDEX(B2, B6, B7+1))
11	's=	=IF(B6=B4+1, 0, INDEX(B2, B6+1, B7))
12	'z=	=3.1415926*INDEX(B2, B6, B7)*B5+(B8+B9+B10+B11)/4

Sheet 54: The NEWESP helper function for computing the esp variable in sheet 53.

We also need a function for computing the initial value of esp given in lines 66 to 68.

	A	B
1	=DEFINE("esp_init", B6, B2, B3, B4, B5)	
2	'rho=	=CONSTARRAY(...)
3	'dx2=	=...
4	'r=	=1
5	'c=	=1
6	'result=	=3.1415926*INDEX(B2, B4, B5)*B3

Sheet 55: The function for calculating the initial value of the esp variable.

We then compute the two acceleration vectors ax and ay. We split the calculations into two separate calls to MAP.

	A	B
1	=DEFINE("ax", B15, B2, B3, B4, B5, B6, B7)	
2	'cell=	=CONSTARRAY(...)
3	'wght=	=CONSTARRAY(...)
4	'esp=	=CONSTARRAY(...)
5	'mass=	=1
6	'q=	=1
7	'k=	=1
8	'i=	=INDEX(INDEX(B2, 1, B7), 1, 1)
9	'j=	=INDEX(INDEX(B2, 1, B7), 1, 2)
10	'xpcell=	=1/4
11	'ypcell=	=1/4
12	'bot=	=(INDEX(B4, B8, B9)-INDEX(B4, B8, B9+1))/B10
13	'top=	=(INDEX(B4, B8+1, B9)-INDEX(B4, B8+1, B9+1))/B10
14	'ex=	=B13*INDEX(INDEX(B3, 1, B7), 1, 4)+B12*INDEX(INDEX(B3, 1, B7), 1, 2)
15	'result=	=B14*B6/B5

Sheet 56: The `AX` function for computing the acceleration in the x-direction.

	A	B
1	=DEFINE("ay", B15, B2, B3, B4, B5, B6, B7)	
2	'cell=	=CONSTARRAY(...)
3	'wght=	=CONSTARRAY(...)
4	'esp=	=CONSTARRAY(...)
5	'mass=	=1
6	'q=	=1
7	'k=	=1
8	'i=	=INDEX(INDEX(B2, 1, B7), 1, 1)
9	'j=	=INDEX(INDEX(B2, 1, B7), 1, 2)
10	'xpcell=	=1/4
11	'ypcell=	=1/4
12	'lft=	=(INDEX(B4, B8, B9)-INDEX(B4, B8+1, B9))/B11
13	'rgt=	=(INDEX(B4, B8, B9+1)-INDEX(B4, B8+1, B9+1))/B11
14	'ey=	=B12*INDEX(INDEX(B3, 1, B7), 1, 3)+B13*INDEX(INDEX(B3, 1, B7), 1, 1)
15	'result=	=B14*B6/B5

Sheet 57: The `AY` function for computing the acceleration in the y-direction.

With the acceleration vectors, we can now calculate the velocity and position vectors. The calculations for the variables `vx1`, `vy1`, `x1` and `y1` are all *daxpy* computations (double-precision $A \cdot X$ plus Y) so we can define a common function for these operations.

	A	B
1	=DEFINE("daxpy", B5, B2, B3, B4)	
2	'a=	=2
3	'x=	=5
4	'y=	=8
5	'result=	=B2*B3+B4

Sheet 58: Function for computing the double-precision variant of $a \cdot x + y$.

We can use this function with `MAP` to compute the vectors, passing in the right arguments in each case. As for the computation for the cell and weight vectors, we also opt to split the computations of `x`, `vx`, `y` and `vy` in lines 109 to 138 to retain the parallel nature of the product-form loop that is used to perform the calculations. We note that the code for the two dimensions are identical, so we only need two functions for all four variables.

	A	B
1	=DEFINE("xy", B5, B2, B3, B4)	
2	'xy_i=	=...
3	'xy_max=	=...
4	'delta=	=1E-10
5	'result=	=IF(B2<0, REFLECT(B2, 0, B3, -B4), IF(B2=0, B2+B4, IF(B2>B3, REFLECT(B2, B3, B3, B4), IF(B2=B3, B2-B4, B2))))

Sheet 59: A function for calculating the `x` and `y` vectors in the particle transport program.

	A	B
1	=DEFINE("vxvy", B5, B2, B3, B4)	
2	'xy1_i=	=...
3	'vxvy1_i=	=...
4	'xy_max=	=1E-10
5	'result=	=IF(OR(B2<0, B2=0, B2>B4, B2=B4), -B3, B3)

Sheet 60: A function for calculating the `vx` and `vy` vectors in the particle transport program.

We finally have all the functions we need to define the entire `move` function that steps forward in time using a fixed delta and updates the position, speed and acceleration of all the particles.

	A	B
1	=DEFINE("move", B34, B2, B3, B4, B5, B6)	
2	'params=	=HCAT(10, 4, 4, 0.1, 1, 1, 1, 1)
3	'xin=	=HCAT(...)

4	'yin=	=HCAT(...)
5	'vxin=	=HCAT(...)
6	'vyin=	=HCAT(...)
7	'np=	=INDEX(B2, 1, 1)
8	'xcell=	=INDEX(B2, 1, 2)
9	'ycell=	=INDEX(B2, 1, 3)
10	'dt=	=INDEX(B2, 1, 4)
11	'q=	=INDEX(B2, 1, 5)
12	'mass=	=INDEX(B2, 1, 6)
13	'xmax=	=INDEX(B2, 1, 7)
14	'ymax=	=INDEX(B2, 1, 8)
15	'xpcell=	=B13/B8
16	'ypcell=	=B14/B9
17	'cell=	=MAP(CLOSURE("cell", NA(), NA(), B15, B16), B3, B4)
18	'wght=	=MAP(CLOSURE("wght", NA(), NA(), B15, B16), B3, B4)
19	'grids=	=GRIDS(B7, B8, B9, B17, B18)
20	'rho=	=RHO(B19)
21	'esp_init=	=TABULATE(CLOSURE("esp_init", B20, B15*B15, NA(), NA()), ROWS(B20), COLUMNS(B20))
22	'esp=	=ESP(1, B15*B15, B21, B8, B9)
23	'ax=	=MAP(CLOSURE("ax", B17, B18, B22, B12, B11, NA(), B15, B16), HSEQ(1, B7, 1)
24	'ay=	=MAP(CLOSURE("ay", B17, B18, B22, B12, B11, NA(), B15, B16), HSEQ(1, B7, 1)
25	'daxpy=	=CLOSURE("DAXPY", NA(), B10, NA())
26	'vx1=	=MAP(B25, B5, B23)
27	'vy1=	=MAP(B25, B4, B24)
28	'x1=	=MAP(B25, B3, B26)
29	'y1=	=MAP(B25, B4, B27)
30	'x=	=MAP(CLOSURE("xy", NA(), B13, 1E-10), B28)
31	'y=	=MAP(CLOSURE("xy", NA(), B14, 1E-10), B29)
32	'vx=	=MAP(CLOSURE("vxvy", NA(), NA(), B13), B28, B26)
33	'vy=	=MAP(CLOSURE("vxvy", NA(), NA(), B14), B29, B27)
34	'result=	=HARRAY(B30, B31, B32, B33)

Sheet 61: The `MOVE` function that updates the positions of the particles.

3.14 Gel Chromatography

```
global LOG(x: double_real returns double_real)

function RUNKUT(COF1, COF2, COF3, COF4, COF5, COF6, RATIO: double_real;
                LN: array[array[double_real]]; N: integer
                returns array[array[double_real]])
    let
        rc1, rc2, rc3, rc4, rc5 :=
            for J in 2, N
                CLI      := LN[1, J];
```

```
10                  CMI      := LN[2, J];
11                  CMLI     := LN[3, J];
12                  CML2I    := LN[4, J];
13                  CML2ISOI := LN[5, J];
14                  RKK1     := COF1 * CMI * CLI + COF2 * CMLI;
15                  RKL1     := -(RKK1 + COF3 * CMLI * CLI + COF4 * CML2I);
16                  RKP1     := RATIO * (RKK1 + RKK1 + RKL1);
17                  RKM1     := COF5 * CML2I + COF6 * CML2ISOI;
18                  RKN1     := -(RKK1 + RKL1 + RKM1);
19                  U        := CLI + 0.5D0 * RKP1;
20                  W        := CML2I + 0.5D0 * RKN1;
21                  XX       := CMLI + 0.5D0 * RKL1;
22                  RKK2     := COF1 * (CMI + 0.5D0 * RKK1) * U + COF2 * XX;
23                  RKL2     := -(RKK2 + COF3 * XX * U + COF4 * W);
24                  RKP2     := RATIO * (RKK2 + RKK2 + RKL2);
25                  RKM2     := COF5 * W + COF6 * (CML2ISOI + 0.5D0 * RKM1);
26                  RKN2     := -(RKK2 + RKL2 + RKM2);
27                  VV       := CLI + 0.5D0 * RKP2;
28                  Y        := CMLI + 0.5D0 * RKL2;
29                  Z        := CML2I + 0.5D0 * RKN2;
30                  RKK3     := COF1 * (CMI + 0.5D0 * RKK2) * VV + COF2 * Y;
31                  RKL3     := -(RKK3 + COF3 * Y * VV + COF4 * Z);
32                  RKP3     := RATIO * (RKK3 + RKK3 + RKL3);
33                  RKM3     := COF5 * Z + COF6 * (CML2ISOI + 0.5D0 * RKM2);
34                  RKN3     := -(RKK3 + RKL3 + RKM3);
35                  R        := CLI + RKP3;
36                  S        := CMLI + RKL3;
37                  T        := CML2I + RKN3;
38                  RKK4     := COF1 * (CMI + RKK3) * R + COF2 * S;
39                  RKL4     := -(RKK4 + COF3 * S * R + COF4 * T);
40                  RKP4     := RATIO * (RKK4 + RKK4 + RKL4);
41                  RKM4     := COF5 * T + COF6 * (CML2ISOI + RKM3);
42                  RKN4     := -(RKK4 + RKL4 + RKM4);
43                  DELK     := (RKK1 + RKK2 + RKK2 + RKK3 + RKK3 + RKK4) / 6.0D0;
44                  DELL     := (RKL1 + RKL2 + RKL2 + RKL3 + RKL3 + RKL4) / 6.0D0;
45                  DELM     := (RKM1 + RKM2 + RKM2 + RKM3 + RKM3 + RKM4) / 6.0D0;
46                  v1       := CLI + RATIO * (DELK + DELK + DELL);
47                  v2       := CMI + DELK;
48                  v3       := CMLI + DELL;
49                  v4       := CML2I - (DELK + DELL + DELM);
50                  v5       := CML2ISOI + DELM;
51              returns array of v1
52                      array of v2
53                      array of v3
54                      array of v4
55                      array of v5
56              end for;
57          r1 := array_addl(rc1, 0.0D0);
58          r2 := array_addl(rc2, 0.0D0);
59          r3 := array_addl(rc3, 0.0D0);
60          r4 := array_addl(rc4, 0.0D0);
61          r5 := array_addl(rc5, 0.0D0);
62      in
63          array [1: r1, r2, r3, r4, r5]
64      end let
65  end function
```

```
66
67  function RENUM(NP: integer;
68                 LN: array[array[double_real]];
69                 N, I, IELUTE: integer;
70                 VSEG: double_real;
71                 CELUTE: array[array[double_real]]
72                 returns array[array[double_real]], double_real)
73      let
74          K         := I / IELUTE;
75          VOL       := double_real(K) * VSEG;
76          CELUTE_1 := CELUTE[1, K: LN[1, N];
77                             2, K: LN[2, N];
78                             3, K: LN[3, N];
79                             4, K: LN[4, N];
80                             5, K: LN[5, N]];
81      in
82          CELUTE_1, VOL
83      end let
84  end function
85
86  function OUT(LN: array[array[double_real]]; N: integer; VOL: double_real; KELUTE:
     ↪   integer;
87               VSEG, F: double_real; CELUTE: array[array[double_real]]
88               returns double_real, array[double_real], array[double_real],
                      ↪   double_real, double_real,
89                     double_real, double_real, double_real, double_real,
                            ↪   double_real, integer,
90                     double_real, double_real, double_real)
91      let CTL,
92          CTM := for J in 1, KELUTE
93                     CTL := CELUTE[1, J] + CELUTE[3, J] + CELUTE[4, J] +
94                            CELUTE[4, J] + CELUTE[5, J] + CELUTE[5, J];
95                     CTM := CELUTE[2, J] + CELUTE[3, J] + CELUTE[4, J] +
96                            CELUTE[5, J];
97                 returns array of CTL
98                         array of CTM
99                 end for;
100         TOTM,
101         TOTML,
102         TOTML2,
103         TOTML2I := for J in 1, KELUTE
104                    returns value of sum CELUTE[2, J]
105                            value of sum CELUTE[3, J]
106                            value of sum CELUTE[4, J]
107                            value of sum CELUTE[5, J]
108                    end for;
109         TOT   := TOTM + TOTML + TOTML2 + TOTML2I;
110         TOTMA := 1.554870369D-05 * 0.486D0;
111         PERML,
112         JSTOR,
113         STOR,
114         PERCENT,
115         HL := if (TOT = 0.0D0) then
116                   0.0D0, 0, 0.0D0, 0.0D0, 0.0D0
117               else
118                   let
```

```
                    PERML := 100.0D0 * (TOTML + TOTML2 + TOTML2I) / TOT;
                    STOR,
                    JSTOR := for initial
                                 STOR  := CTL[915];
                                 JSTOR := 915;
                                 J     := 915;
                             while (J <= 1190) repeat
                                 J := old J + 1;
                                 STOR,
                                 JSTOR := if (old STOR < CTL[old J]) then
                                              old STOR, old JSTOR
                                          else
                                              CTL[old J], old J
                                          end if;
                             returns value of STOR
                                     value of JSTOR
                             end for;
                    TOT1 := for J in 1, KELUTE
                            returns value of sum CTL[J]
                            end for;
                    TOT2 := for J in 1, JSTOR
                            returns value of sum CTL[J]
                            end for;
                    PERCENT := 100.0D0 * TOT2 / TOT1;
                    HL      := -LOG(2.0D0) * double_real(JSTOR) *
                               0.016D0 / (F * LOG(PERCENT / 18.02233D0));
                in
                    PERML, JSTOR, STOR, PERCENT, HL
                end let
            end if;
    in
        VOL, CTM, CTL, TOTM, TOTML, TOTML2, TOTML2I, TOT, PERML, TOTMA,
        JSTOR, STOR, PERCENT, HL
    end let
end function

function FILLUP(N: integer; DX: double_real; KELUTE: integer; VSEG: double_real;
                NSEG: integer; GZERO: array[double_real]
                returns array[double_real], array[array[double_real]],
                        array[array[double_real]], array[double_real])
    let
        X_c := for J in 2, N
               returns array of double_real(J - 1) * DX
               end for;
        X   := array_addl(X_c, 0.0D0);
        V   := for J in 1, KELUTE
               returns array of double_real(J) * VSEG
               end for;
        C   := for M in 1, 5
                   Cr := array_fill(2, NSEG, GZERO[M]) ||
                         array_fill(NSEG + 1, N, 0.0D0);
               returns array of array_addl(Cr, 0.0D0)
               end for;
        CELUTE := for I in 1, 5 cross J in 1, KELUTE
                  returns array of 0.0D0
                  end for;
```

```
175     in
176         X, C, CELUTE, V
177     end let
178 end function
```

Listing 23: The gel chromatography program for simulating observed elution patterns of proteins and ligands in a column of gel.

Like the particle transport program, the gel chromatography program consists only of arithmetic and loops. Due to the lenghty functions in this program, we have opted to move the SDFs to a separate appendix for the interested reader. We begin by defining the `RUNKUT` function.

	A	B
1	=DEFINE("runkut", B17, B2, B3, B4, B5, B6, B7, B8, B9)	
2	'cof1=	=1
3	'cof2=	=2
4	'cof3=	=3
5	'cof4=	=4
6	'cof5=	=5
7	'cof6=	=6
8	'ratio=	=0.0005
9	'ln=	=VARRAY(...)
10	'n=	=3
11	'?=	=RUNKUT_HELPER(2, B10, HARRAY(B2, B3, B4, B5, B6, B7), B9, B8, CONSTARRAY(HARRAY(), 1, 5))
12	'rc1=	=INDEX(B11, 1, 1)
13	'rc2=	=INDEX(B11, 1, 2)
14	'rc3=	=INDEX(B11, 1, 3)
15	'rc4=	=INDEX(B11, 1, 4)
16	'rc5=	=INDEX(B11, 1, 5)
17	'result=	=HARRAY(HCAT(0, B12), HCAT(0, B13), HCAT(0, B14), HCAT(0, B15), HCAT(0, B16))

Sheet 62: The `RUNKUT` function from the SISAL gel chromatography program in Funcalc.

In cell `B11` of sheet 62, we use a helper function for calculating the many computations given in lines 9–50. This function is given below.

	A	B
1	=DEFINE("runkut_helper", B61, B2, B3, B4, B5, B6, B7)	
2	'j=	=2
3	'n=	=4
4	'coefficients=	=HARRAY(...)
5	'ln=	=VARRAY(...)
6	'ratio=	=0.0005

7	'v_accs=	=HARRAY(HARRAY(), ...)
8	'cof1=	=INDEX(B4, 1, 1)
9	'cof2=	=INDEX(B4, 1, 2)
10	'cof3=	=INDEX(B4, 1, 3)
11	'cof4=	=INDEX(B4, 1, 4)
12	'cof5=	=INDEX(B4, 1, 5)
13	'cof6=	=INDEX(B4, 1, 6)
14	'cli=	=INDEX(INDEX(B5, 1, 1), 1, B2)
15	'cmi=	=INDEX(INDEX(B5, 2, 1), 1, B2)
16	'cmli=	=INDEX(INDEX(B5, 3, 1), 1, B2)
17	'cml2i=	=INDEX(INDEX(B5, 4, 1), 1, B2)
18	'cml2isoi=	=INDEX(INDEX(B5, 5, 1), 1, B2)
19	'rkk1=	=B8*B15*B14+B9*B16
20	'rkl1=	=-(B19+B10*B16*B14+B14*B17)
21	'rkp1=	=B6*(B19+B19+B20)
22	'rkm1=	=B12*B17+B13*B18
23	'rkn1=	=-(B19+B20+B22)
24	'u=	=B14+0.5*B21
25	'w=	=B17+0.5*B23
26	'xx=	=B16+0.5*B20
27	'rkk2=	=B8*(B15+0.5*B19)*B24+B9*B26
28	'rkl2=	=-(B27+B10*B26*B24+B11*B25)
29	'rkp2=	=B6*(B27+B27+B28)
30	'rkm2=	=B12*B25+B13*(B18+0.5*B22)
31	'rkn2=	=-(B27+B28+B30)
32	'vv=	=B14+0.5*B29
33	'y=	=B16+0.5*B28
34	'z=	=B17+0.5*B31
35	'rkk3=	=B8*(B15+0.5*B27)*B32+B9*B33
36	'rkl3=	=-(B35+B10*B33*B32+B11*B34)
37	'rkp3=	=B6*(B35+B35+B36)
38	'rkm3=	=B12*B34+B13*(B18+0.5*B30)
39	'rkn3=	=-(B35+B36+B38)
40	'r=	=B14+B37
41	's=	=B16+B36
42	't=	=B17+B39
43	'rkk4=	=B8*(B15+B35)*B40+B9*B41
44	'rkl4=	=-(B43+B10*B41*B40+B11*B42)
45	'rkp4=	=B6*(B43+B43+B44)
46	'rkm4=	=B12*B42+B13*(B18+B38)
47	'rkn4=	=-(B43+B44+B46)
48	'delk=	=(B19+B27+B27+B35+B35+B43)/6.0
49	'dell=	=(B20+B28+B28+B36+B36+B44)/6.0
50	'delm=	=(B22+B30+B30+B38+B38+B46)/6.0
51	'v1=	=B14+B6*(B48+B48+B49)
52	'v2=	=B15+B48
53	'v3=	=B16+B49
54	'v4=	=B17-(B48+B49+B50)
55	'v5=	=B18+B50
56	'v1_acc=	=INDEX(B7, 1, 1)
57	'v2_acc=	=INDEX(B7, 1, 2)
58	'v3_acc=	=INDEX(B7, 1, 3)
59	'v4_acc=	=INDEX(B7, 1, 4)

60	'v5_acc=	=INDEX(B7, 1, 5)
61	'result=	=IF(B2>B3, HARRAY(B56, B57, B58, B59, B60), RUNKUT_HELPER(B2+1, B3, B4, B5, B6, HARRAY(HCAT(B56, B51), HCAT(B57, B52), HCAT(B58, B53), HCAT(B59, B54), HCAT(B60, B55))))

Sheet 63: The recursive `RUNKUT_HELPER` SDF which calculates the long list of computations in lines 9–50.

	A	B
1	=DEFINE("renum", B11, B2, B3, B4, B5, B6, B7)	
2	'ln=	=VARRAY(...)
3	'n=	=...
4	'i=	=...
5	'ielute=	=...
6	'vseg=	=...
7	'celute=	=CONSTARRAY(...)
8	'k=	=MAX(FLOOR(B4/B5, 1)
9	'vol=	=B8*B6
10	'celute_1=	=UPDATEARRAY(B7, SLICE(B2, 1, B3, ROWS(B2), B3), 1, B8)
11	'result=	=HARRAY(B10, B9)

Sheet 64: The `RENUM` function in Funcalc. See appendix A for the definition of `UPDATEARRAY`.

	A	B
1	=DEFINE("fillup", B18, B2, B3, B4, B5, B6, B7)	
2	'n=	=...
3	'dx=	=...
4	'kelute=	=...
5	'vseg=	=...
6	'nseg=	=...
7	'gzero=	=...
8	'x_c=	=TABULATE(CLOSURE("j_dx", B3, NA(), NA()), 1, B2-1)
9	'x=	=HCAT(0, B8)
10	'v=	=TABULATE(CLOSURE("j_vseg", B5, NA(), NA()), 1, B4)
11	'ctemp1=	=HCAT(0, CONSTARRAY(INDEX(B7, 1, 1), 1, B6-2+1), CONSTARRAY(0, 1, B6+1))
12	'ctemp2=	=HCAT(0, CONSTARRAY(INDEX(B7, 1, 2), 1, B6-2+1), CONSTARRAY(0, 1, B6+1))
13	'ctemp3=	=HCAT(0, CONSTARRAY(INDEX(B7, 1, 3), 1, B6-2+1), CONSTARRAY(0, 1, B6+1))
14	'ctemp4=	=HCAT(0, CONSTARRAY(INDEX(B7, 1, 4), 1, B6-2+1), CONSTARRAY(0, 1, B6+1))
15	'ctemp5=	=HCAT(0, CONSTARRAY(INDEX(B7, 1, 5), 1, B6-2+1), CONSTARRAY(0, 1, B6+1))
16	'c=	=HARRAY(B11, B12, B13, B14, B15)
17	'celute=	=CONSTARRAY(0, 5, B4)
18	'result=	=HARRAY(B9, B16, B17, B10)

Sheet 65: The FILLUP SDF in Funcalc.

We now define the auxiliary functions j_dx and j_vseg that calculate the multiplication of variables v and x_c in lines 160 to 166.

	A	B
1	=DEFINE("j_dx", B5, B2, B3, B4)	
2	'dx=	=...
3	'r=	=...
4	'c=	=...
5	'result=	=B4*B2

Sheet 66: The J_DX auxiliary function.

	A	B
1	=DEFINE("j_vseg", B5, B2, B3, B4)	
2	'dx=	=82.824
3	'r=	=1
4	'c=	=1
5	'result=	=B4*B2

Sheet 67: The J_VSEG auxiliary function.

	A	B
1	=DEFINE("out", B24, B2, B3, B4, B5, B6, B7)	
2	'ln=	=VARRAY(...)
3	'n=	=3
4	'vol=	=4.56465
5	'kelute=	=1200
6	'vseg=	=7.42524
7	'f=	=1.4254
8	'celute=	=CONSTARRAY(...)
9	'ctl=	=CTL_HELPER(1, B5, B8, HARRAY())
10	'ctm=	=CTM_HELPER(1, B5, B8, HARRAY())
11	'totm=	=SUM(SLICE(B8, 2, 1, 2, B5))
12	'totml=	=SUM(SLICE(B8, 3, 1, 3, B5))
13	'totml2=	=SUM(SLICE(B8, 4, 1, 4, B5))
14	'totml2i=	=SUM(SLICE(B8, 5, 1, 5, B5))
15	'tot=	=B11+B12+B13+B14
16	'totma=	=1.554870369E-05*0.486
17	'perml=	=IF(B15=0, 0, 100*(B12+B13+B14)/B15)
18	'jstor=	=IF(B15=0, 0, INDEXMAX(SLICE(B9, 1, 915, 1, 1190))+914)
19	'stor=	=IF(B15=0, 0, INDEX(B9, 1, B17))
20	'percent=	=IF(B15=0, 0, 100*B22/B21)
21	'hl=	=IF(B15=0, 0, -LOG10(2)*B17*0.016/ (B7*LOG10(B19/18.02233)))
22	'tot1=	=SUM(B9)
23	'tot2=	=SUM(SLICE(B9, 1, 1, 1, B17))
24	'result=	=HARRAY(B4, B10, B9, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21)

Sheet 68: The OUT main SDF in the gel chromatography program.

Finally, we define the two helper functions that we use to calculate ctl and ctm variables in cells B9 and B10 of sheet 68. Note that we use the INDEXMAX function which does the same as INDEXMIN from section 3.6 but finds the index for the greatest element.

	A	B
1	=DEFINE("ctl_helper", B6, B2, B3, B4, B5)	
2	'j=	=1
3	'kelute=	=5
4	'celute=	=CONSTARRAY(...)
5	'acc=	=HARRAY()
6	'result=	=IF(B2<=B3, CTL_HELPER(B2+1, B3, B4, HCAT(B5, INDEX(B4, 1, B2)+INDEX(B4, 3, B2)+INDEX(B4, 4, B2)*2+INDEX(B4, 5, B2))), B5)

Sheet 69: The CTL_HELPER helper function.

	A	B
1	`=DEFINE("ctm_helper", B6, B2, B3, B4, B5)`	
2	`'j=`	`=1`
3	`'kelute=`	`=5`
4	`'celute=`	`=CONSTARRAY(...)`
5	`'acc=`	`=HARRAY()`
6	`'result=`	`=IF(B2<=B3, CTM_HELPER(B2+1, B3, B4, HCAT(B5, INDEX(B4, 2, B2)+INDEX(B4, 3, B2)+INDEX(B4, 4, B2)+INDEX(B4, 5, B2))), B5)`

Sheet 70: The `CTM_HELPER` helper function.

4 Conclusion and Future Work

We conclude this report by discussing and summarising the expressiveness of Funcalc and suggesting directions for future work.

4.1 Funcalc Expressiveness

We can safely conclude that Funcalc is a very expressive language despite the relatively small number of built-in functions available in the current version as of this writing (last update on October 5th, 2016) and its lack of some of the benefits of contemporary functional languages, which is largely remedied by SDFs. In only a small number of cases did we need to define our own utility functions in Funcalc such as `UPDATEARRAY` (see subappendix A.4). One reason is perhaps that both SISAL and Funcalc were developed to handle computation while having different computational paradigms: SISAL was built for high-performance scientific computing as a functional replacement or alternative for the then dominant Fortran language, while Funcalc was built predominantly to demonstrate that SDFs can be efficient and expressive tools for computations in spreadsheets. In the next paragraphs, we briefly reiterate cases where Funcalc could elegantly express SISAL programs and some of the limitations we encountered.

Records and unions were not readily expressible in Funcalc without some trickery. As described in section 2.1.4 and section 2.1.5, we can emulate both of the structures using arrays of data and some supporting SDFs, but we are sceptical about their usefulness in real-world spreadsheets. Streams were not expressible either because Funcalc is eager, but we could perhaps use closures as thunks to delay computation. This is not likely a useful tool for end-users in general. Programs like the Sieve of Eratosthenes would benefit

greatly from laziness to avoid generate huge arrays where a large part of them are subsequently filtered away. Although we encountered some initial problems with manipulating arrays, like array concatenation in section 3.11 and updating part of an array in section 3.13, we managed to express these constructs anyway, albeit in a less efficient manner.

Still one cannot help but envy the conveniences of modern functional programming languages. Suggestions for some of these limitations are given in the next section and attempt to take into consideration that the target audience of Funcalc are end-users that usually do not have any formal computer science training, so we cannot simply implement sophisticated features from the functional programming world without taking this into account [10].

4.2 Directions For Future Work

The following bullet points summarise suggestions for future work that were conceived during the writing of this report, and in particular the SDFs of section 3.

- Funcalc would benefit from functions to traverse single-row or single-column arrays like lists in functional languages. At the moment one can use recursion as a looping construct, but this presents some problems:

  1. Recursively shortening the array using slicing requires a reliable, and preferably fast, way of testing if the array is empty which is also invariant to the orientation of the input. While we can use `ROWS` and `COLUMNS` as we did in section 3.7, we still need to know beforehand if we are working with a row or a column, although we could check both. An `EMPTY` function can be defined as a SDF (see appendix A)

  2. Passing the array directly and using an index is fine, but requires two separate functions for the horizontal and vertical directions, or some of trickery to have a single function work for both directions.

  3. One can just use `TRANSPOSE` for creating the vertical or horizontal counterpart, but this requires either that the user remembers to use `TRANSPOSE` or that we define two functions for every case: One "real" function and a transposed counterpart for the other

direction which will take a performance hit.

- Readability would be improved if instead of `CLOSURE("MY_FUNC")`, we could instead simply write `MY_FUNC` or `"MY_FUNC"`. The type system could ensure that the syntax desugaring only happens when the argument is expected to be a function value.
- `ROWMAP` and `COLMAP` should accept a function value that takes a single row/column as argument for scalability and maintainability, instead of a function that takes as many arguments as there are elements in the row or column. In their current state, they require end-users to know the exact number of arguments for the function. We also sacrifice generality as a given function can only be used for a specific number of arguments. The updated functions would be able to express the same as what they can express in their current state.
- Anonymous function closures would be very beneficial in Funcalc. As an example, take the function `=MAP(CLOSURE("MULT3_ADD2_COS"), array)` which for each element multiplies by 3, adds 2 and then applies the cosine function to the result. Normally one would need to define a new SDF which may only be used once. We encountered this multiple times when translating the SISAL programs. Instead, one could use some hypothetical syntax for anonymous functions, e.g. `=MAP(COS(@1 * 3 + 2), array)`, where `@1` refers to the elements in the input array. For multiple arguments, one would use `@2`, `@3`, etc. while `@*` could perhaps refer to all the arguments packaged as an array. Notice that we are assuming that we can omit the `CLOSURE` function as previously described. The anonymous function should be cached with its expression for reuse when a similar expression is given for an argument that requires a function value. Current work is being conducted in order to identity this proposal's viability and possible challenges.
- We have repeatedly encountered the following pattern: Perform some computations for some number `i` until some other number `n` where `i < n` incrementing `i` after each round of computation. The solution has been a recursive function, but perhaps more elegant solutions exist such as a function taking a predicate which is tested on each iteration.
- There are currently no tools for debugging in general or debugging recursive SDFs. This makes the process cumbersome and error-prone. As evident from section 3, Funcalc programs with several intricate loops can quickly become unwieldy requiring extra care from the user

to ensure correctness and avoid infinite loops, even if infinite recursion was guarded. For example, a debugging tool for stepping through the execution of an SDF would be very useful.

- The highlighting of intermediate calculations in a SDF helped uncover unused variables. In the `RUNKUT_HELPER` function, variables `RKN4`, `RKP4` and `RKN4` are not used and in the `RENUM` function in the gel chromatography program (section 3.14), the parameter `NP` is not used either (in fact, the `RENUM` function itself is not even used).
- The generalised `MAP` function is very useful as it also acts as a zip function for multiple arguments. The function argument to `MAP` must accept as many arguments as there are arrays in the subsequent arguments. This requirement degrades the general practicality of the `MAP` function and indeed any other function that uses the same approach like `ROWMAP` and `COLMAP`. Consider using `MAP` for computing some values of n arrays. We assume the existence of a function `FUNC` that takes exactly n arguments. The call then becomes `MAP(CLOSURE("FUNC"),` $\texttt{array}_1$`, ...,` $\texttt{array}_n$`)`. Should n ever need to change, we must define a new function that takes that number of arguments instead, even though the two functions do the same form of computation. Clearly, this is not scalable. Instead, one solution would be that `MAP` passes an array of n arguments to the function closure. This does require that the function value can handle an arbitrary number of arguments. Functions that need to operate on individual elements are still possible since we can extract the elements of the array using `INDEX` at the expense of multiple calls to `INDEX` that may worsen readability.
- Array formulas would be useful tools in SDFs. They would allow for result unpacking as explained in section 2.4, however using arrays and indexing works as well.
- It would interesting to find a minimal set of functions that can implement all the functions in Excel for example, or to verify whether the current set of intrinsic functions in Funcalc is minimal in this regard.
- Similar to how one can view all formulas in a spreadsheet using a formula view, it would very convenient to have a *variable view* for function sheets where cell references are replaced by the variable names that appear on the left side of the computations of a SDF. This would greatly aid debugging as it is easier to refer to named variables than cell references which can be a source of off-by-one errors where an incorrect

cell is referenced. It can also lead to infinite loops. In this report, we have kept variable names on the left side of the values, but this was just a convention we chose arbitrarily. Others may feel it is more natural to use a horizontal layout where the variable names appear above the values. This complicates finding the mapping between variable names and values, and more work is needed to find strategies for making the view as reliable as the formula view.

- We often encountered a situation where we needed to compute a value by having a initial value then passing it to a function. The result of that function application is then given as input to the same functions and so on as in $f^3(x) = f(f(f(x)))$. We cannot use `MAP` in this case as the each computation depends on the previous computation. Instead, this could be abstracted into a recursive function called `SEQUENCE` or `CHAIN` that applies a function to a starting value n times. This would obviate the need for continuously defining recursive functions which do the same computations but with different functions. An implementation is given in subappendix A.8.

References

[1] David C. Cann. *SISAL 1.2: A Brief Introduction and Tutorial*. Lawrence Livermore National Laboratory.

[2] Peter Sestoft. "Corecalc and Funcalc Spreadsheet Technology in C#". In: (2014). URL: http://www.itu.dk/people/sestoft/funcalc/ (visited on 10/12/2016).

[3] Peter Sestoft. *Spreadsheet Implementation Technology*. The MIT Press, 2014. ISBN: 9780262526647.

[4] James McGraw et al. *SISAL: Streams and Iteration in a Single Assignment Language, Language Referece Manual*. Tech. rep. Version 1.2. Lawrence Livermore National Laboratory, Mar. 1, 1985.

[5] D. C. Cann et al. *SISAL Reference Manual: Language Version 2.0*. Lawrence Livermore National Laboratory, 1992.

[6] Victor N. Kasyanov and Alexander P. Stasenko. "A Functional Programming System SFP: Sisal 3.1 Language Structures Decomposition". In: *Parallel Computing Technologies: 9th International Conference, PaCT 2007, Pereslavl-Zalessky, Russia, September 3-7, 2007. Proceedings*. Ed. by Victor Malyshkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 62–73. ISBN: 978-3-540-73940-1. DOI: 10.1007/978-3-540-73940-1_6. URL: http://dx.doi.org/10.1007/978-3-540-73940-1_6.

[7] Project CROAP. *Sisal*. URL: http://www-sop.inria.fr/croap/transllprog/subsection3_3_3.html (visited on 11/02/2016).

[8] J. Hughes. "Why Functional Programming Matters". In: *Comput. J.* 32.2 (Apr. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. URL: http://dx.doi.org/10.1093/comjnl/32.2.98.

[9] Zhenjiang Hu, John Hughes, and Meng Wang. "How Functional Programming Mattered". In: *National Science Review* (2015). DOI: 10.1093/nsr/nwv042. eprint: http://nsr.oxfordjournals.org/content/early/2015/07/13/nsr.nwv042.full.pdf+html. URL: http://nsr.oxfordjournals.org/content/early/2015/07/13/nsr.nwv042.abstract.

[10] Simon Peyton-Jones, Alan Blackwell, and Margaret Burnett. "A User-centred Approach to Functions in Excel". In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM, 2003, pp. 165–176. ISBN: 1-58113-756-7. DOI: 10.1145/944705.944721. URL: http://doi.acm.org/10.1145/944705.944721.

List of Spreadsheets

List of Tables

List of Listings

Appendices

Appendix A: Auxiliary Functions

This appendix contains the Funcalc auxiliary helper SDFs used in some of the example programs in section 3.

A.1: IMAP

	A	B
1	=DEFINE("imap", B4, B2, B3)	
2	'fv=	=...
3	'array=	=...
4	'result=	=MAP(B3, HSEQ())

Sheet 71: An extended version of the MAP function which also passes the one-based index of the current element to the argument closure which has to be a binary function, accepting the index as its first argument and the element as the second. As it is not possible to specify variadic functions in Funcalc without constructing a built-in function, IMAP only maps over a single array.

A.2: Tail-Recursive Factorial Function

	A	B
1	=DEFINE("factorial", B4, B2, B3)	
2	'n=	=...
3	'acc=	=...
4	'result=	=IF(B2=0, B3, FACTORIAL(B2-1, B2*B3))

Sheet 72: More efficient, tail-recursive factorial function.

A.3: SUMPRODUCT

	A	B
1	=DEFINE("sumproduct", B4, B2, B3)	
2	'array1=	=HARRAY(1, 2, 3)
3	'array2=	=HARRAY(4, 5, 6)
4	'out=	=SUM(MAP(*, B2, B3))

Sheet 73: Defining the SUMPRODUCT SDF. Here, we assume we can bind built-in functions to closures. In the real implementation, we define a PRODUCT SDF that is passed to MAP instead.

A.4: UPDATEARRAY

	A	B
1	=DEFINE("assign", B8, B2, B3, B4, B5, B6, B7)	
2	'array=	=...
3	'subarray=	=...
4	'tr=	=2
5	'tc=	=2
6	'r=	=1
7	'c=	=1
8	'result=	=IF(AND(B6>=B4, B7>=B5, B6<B4+ROWS(B3), B7<B5+COLUMNS(B3)), INDEX(B3, B6-B4+1, B7-B5+1), INDEX(B2, B6, B7))

Sheet 74: Helper function for updating part of an array.

	A	B
1	=DEFINE("updatearray", B6, B2, B3, B4, B5)	
2	'array=	=...
3	'subarray=	=...
4	'r=	=2
5	'c=	=2
6	'result=	=TABULATE(CLOSURE("assign", B2, B3, B4, B5, NA(), NA()), ROWS(B2), COLUMNS(B2))

Sheet 75: SDF for returning a modified copy of an existing array.

A.5: HSEQ

HSEQ takes a start value, an end value and a step value and returns an array of the all the values starting from the start value to the end value in increments of the step value. Calling HSEQ with the placeholder values in sheet 76 will yield the array =HARRAY(0, 2, 4, 6, 8, 10). It is straight-forward to implement a tail-recursive variant of HSEQ and a vertical counterpart VSEQ.

	A	B
1	=DEFINE("hseq", B6, B2, B3, B4, B5)	
2	'z=	=0
3	'n=	=10
4	'step=	=2
5	'result=	=IF(B2>B3, HARRAY(), HCAT(B2, HSEQ(B2+B4, B3, B4)))

Sheet 76: A recursive SDF for creating a range of values in an interval with a given step value.

A.6: ISCHAR and INDEXAT

	A	B
1	=DEFINE("ischar", B3, B2)	
2	'char=	="C"
3	'result=	=NOT(EXTERN("System.Char.IsWhiteSpace$(C)Z", B2))

Sheet 77: Test if a character is whitespace or not by calling C#.

	A	B
1	=DEFINE("indexat", B3, B2)	
2	'string=	="This is a string"
3	'index=	=5
4	'result=	=EXTERN("System.String.Substring(II)T", B2, B3-1, 1)

Sheet 78: Indexing a string in Funcalc using C#. Notice that we subtract one from the index when calling the external function in order to keep indexing one-based.

Notice that we are using System.String.Substring as it does not seem possible to call the index operator string[i] from Funcalc, nor could we successfully call the EnumerableAt extension method.

A.7: GENERATE

	A	B
1	=DEFINE("gen_at", B5, B2, B3, B4)	
2	'fv=	=CLOSURE("...")
3	'r=	=1
4	'c=	=1
5	'result=	=APPLY(B2)

Sheet 79: The helper function for calling a closure at each position in an array created by `TABULATE`.

	A	B
1	=DEFINE("generate", B5, B2, B3, B4)	
2	'fv=	=CLOSURE(...)
3	'rows=	=3
4	'cols=	=3
5	'result=	=TABULATE(CLOSURE("GEN_AT", B2, NA(), NA()), B3, B4)

Sheet 80: A SDF for generating an array of a given size using a nullary function. We cannot use `CONSTARRAY` since it evaluates its argument only once.

A.8: SEQUENCE/CHAIN

	A	B
1	=DEFINE("sequence", B5, B2, B3, B4)	
2	'fv=	=CLOSURE(...)
3	'z=	=1
4	'n=	=10
5	'result=	=IF(B4>0, SEQUENCE(B2, APPLY(B2, B3), B4-1), B3)

Sheet 81: A function for applying a function n times to an initial value. Alternative names might have been `CHAIN` or `APPLYN`.